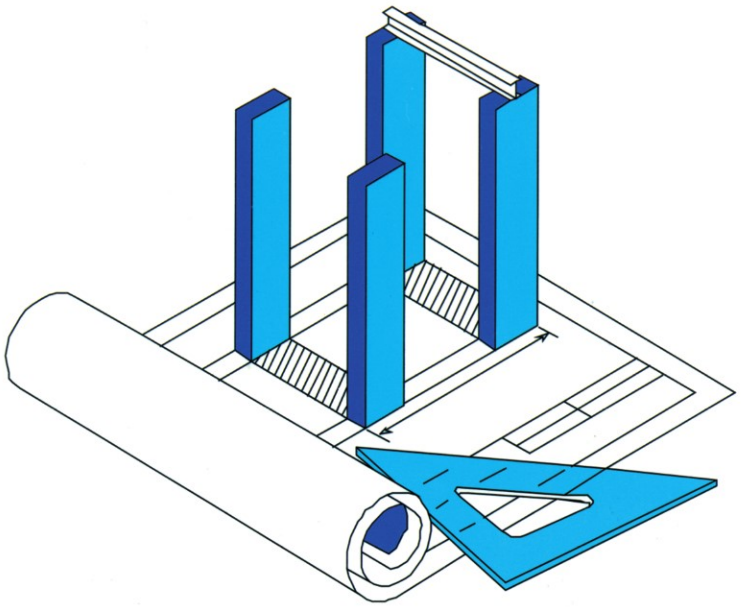
SOFTWARE ARCHITECTURES AND COMPONENT TECHNOLOGY



edited by

Mehmet Akşit

SPRINGER SCIENCE+BUSINESS MEDIA, LLC

SOFTWARE ARCHITECTURES AND COMPONENT TECHNOLOGY

THE KLUWER INTERNATIONAL SERIES
IN ENGINEERING AND COMPUTER SCIENCE

SOFTWARE ARCHITECTURES AND COMPONENT TECHNOLOGY

edited by

Mehmet Akşit
University of Twente, The Netherlands

SPRINGER SCIENCE+BUSINESS MEDIA, LLC

ISBN 978-1-4613-5286-0 ISBN 978-1-4615-0883-0 (eBook)
DOI 10.1007/978-1-4615-0883-0

Library of Congress Cataloging-in-Publication Data

A C.I.P. Catalogue record for this book is available
from the Library of Congress.

The image used on the front cover was obtained from IMSI's MasterClips(R) and MasterPhotosTM Premium Image Collection, 1895 Francisco Blvd. East, San Rafael, CA 94901-5506, USA

Contents

Contributors

The following authors contributed to this book:

Franz Achermann
Mehmet Akşit
Pierre America
Lodewijk Bergmans
Theo D'Hondt
Patrick Th. Eugster
Rachid Guerraoui
Dieter K. Hammer
Kim Mens
Mira Mezini
Oscar Nierstrasz
Henk Obbink
Rob van Ommering
Harold Ossher
Linda Seiter
Joe Sventek
Peri Tarr
Bedir Tekinerdoğan
Kris De Volder
Jan Gerben Wijnstra
Roel Wuyts
Karl Lieberherr

Acknowledgements

Many people contributed towards the creation of this volume. First of all, I would like to express my deepest appreciation to the authors of the chapters and the referees for providing this excellent material. Secondly, I would like to thank the members of the TRESE group, and in particular Lodewijk Bergmans, Klaas van den Berg, Pim van den Broek, Maurice Glandrup, Arend Rensink and Richard van de Stadt for helping me in editing and formatting the volume. Last but not least, without the endless support of Lance Wobus and Sharon Palleschi from Kluwer, finishing this volume would not have been possible.

This work has been partially supported and funded by various organizations including Siemens-Nixdorf Software Center, the Dutch Ministry of Economical affairs under the SENTER program, the Dutch Organization for Scientific Research (NWO, 'Inconsistency management in the requirements analysis phase' project), the AMIDST project, and by the IST Project 1999-14191 EASYCOMP.

Mehmet Akşit

Preface

Software architectures have gained a wide popularity in the last decade and they are generally considered to play a fundamental role in coping with the inherent difficulties of the development of large-scale and complex software systems. Software architectures include the early design decisions and embody the overall structure that impacts the quality of the whole system. A common assumption is that architecture design should support the required software system qualities such as robustness, adaptability, reusability and maintainability.

Component-oriented programming enables software engineers to implement complex applications from a set of pre-defined components. Component-oriented programming is becoming more and more popular probably because it offers a compromise between custom-made software, i.e., software that is developed from scratch and standard software, i.e., prefabricated complete solutions that can only be parameterized to get close enough to what is needed in a particular scenario.

This book collects excellent chapters on software architectures and component technologies from well-known authors, who do not only explain the advantages but also present the shortcomings of the current approaches and introduce novel solutions to overcome the shortcomings.

The first two chapters are introductory of nature; they give definitions, compare various approaches and identify the obstacles that designers may experience while applying architecture and component technologies. The first chapter provides a classification and evaluation of software architecture design methods. For this, contemporary definitions on software architectures are analyzed and a general definition of software architecture is introduced. The problems of various architecture design methods are described. The

second chapter provides a set of guidelines to identify the obstacles that software engineers may experience while designing distributed systems using the current component technology. To this aim, the computer science domain is divided into several sub-domains, and each sub-domain is described using its important aspects. Further, each aspect is analyzed with respect to the current component technology. This analysis helps software engineers to identify the possible obstacles for each aspect of a sub-domain. These two chapters conclude with references to the relevant research activities that are presented in this book.

The chapters 3 to 8 present various architecture design approaches with a strong emphasis on component technology. The chapters 3, 4 and 5 present architecture-design methods exemplified with industrial applications.

In chapter 3 written by Dieter Hammer, first overview of architecture design dimensions and views are presented. The second part of this chapter summarizes the requirements that components must fulfill in order to be composable in the context of dependable distributed real-time systems. Finally, a method for constructing the collective behavior of a set of components and achieving composability is sketched and demonstrated by means of an example.

Chapter 4, written by Henk Obbink, Rob van Ommering, Jan Gerben Wijnstra and Pierre America, explains how component oriented product family architectures provide a promising architecture development paradigm. This paradigm solves the inherent dilemma of the need for careful engineering versus rapid realisation of a large variety of product instances. The approach is illustrated using examples from the medical and the consumer domain.

In chapter 5, Bedir Tekinerdoğan and Mehmet Akşit present a synthesis-based architecture design approach (Synbad). In this method, the client's perspective is abstracted to derive the technical problems. The technical problems define the scope of the solution domains from which the architectural abstractions are derived. The approach is illustrated for the design of an atomic transaction architecture for a real industrial project.

In chapter 6, Patrick Eugster, Rachid Guerraoui and Joe Sventek present the so-called Distributed Asynchronous Collections (DACs) as a set of stable and useful component abstractions with the context of distributed system architecture. By viewing the elements of our DACs as events, these collections can be seen as programming abstractions for asynchronous distributed interaction, enabling the loose coupling of components.

The chapters 7 and 8 can be considered as a bridge between architecture specifications and software components.

In chapter 7, Theo D'Hondt, Kris De Volder, Kim Mens and Roel Wuyts, present a number of experiments based on logic meta-programming to

augment an implementation with enforceable design concerns, including architectural concerns. This approach can be used to codify design information as constraints or even as a process for code generation.

In chapter 8, Mehmet Akşit and Bedir Tekinerdoğan introduce a technique to depict, compare and select among the design alternatives, based on their adaptability and time performance factors. This technique is formally specified and implemented by a number of tools.

The chapters 9 to 12 tackle the problems of current component-based approaches. Each chapter presents a new approach along this line.

Chapter 9, written by Franz Achermann and Oscar Nierstrasz, introduces the programming language Piccola, which is suitable for composing applications from software components. It has a small syntax and a minimal set of features needed for specifying different styles of software composition. Through a series of examples, this chapter illustrates how Piccola suffice to express a variety of compositional abstractions and styles.

Chapter 10, written by Harold Ossher and Peri Tarr, claims that most languages and modularization approaches support only one "dominant" kind of modularization. Once a system has been decomposed, extensive refactoring and reengineering are needed to remodularize it. This chapter presents *hyperspaces* and Hyper/J™ as a particular approach to providing multi-dimensional separation of concerns.

Chapter 11, written by Mira Mezini, Linda Seiter and Karl Lieberherr, addresses object-oriented component integration issues. The chapter argues that traditional framework customization techniques are inappropriate for component-based programming since they lack support for non-invasive, encapsulated, dynamic customization. The chapter proposes a language construct, called a *pluggable composite adapter* for expressing component gluing for better better modularity, flexible extensibility, and improved maintenance and understandability.

Chapter 12, written by Lodewijk Bergmans, Mehmet Akşit and Bedir Tekinerdoğan, first discusses a number of software reuse and extension problems in current object-oriented languages. A number of examples illustrate that both inheritance and aggregation mechanisms cannot adequately express certain aspects of evolving software. As a solution to these problems, the composition filters model is introduced. This chapter also evaluates the effectiveness of various language mechanisms in coping with evolving software as in the presented change case.

PART 1

INTRODUCTION AND OVERVIEW

Chapter 1

CLASSIFYING AND EVALUATING ARCHITECTURE DESIGN METHODS

Bedir Tekinerdoğan and Mehmet Akşit
TRESE Group, Department of Computer Science, University of Twente, postbox 217, 7500 AE, Enschede, The Netherlands. email: {bedir, aksit}@cs.utwente.nl, www: http://trese.cs.utwente.nl

Keywords: Software architecture, classification of architecture design methods, problems in designing architectures

Abstract: The concept of software architecture has gained a wide popularity and is generally considered to play a fundamental role in coping with the inherent difficulties of the development of large-scale and complex software systems. This chapter provides a classification and evaluation of existing software architecture design methods. For this, contemporary definitions on software architectures are analyzed and a general definition of software architecture is introduced. Further, a meta-model for architecture design methods is presented, which is used as a basis for comparing various architecture design approaches. The problems of each architecture design method are described and the corresponding conclusions are given.

1. INTRODUCTION

Software architectures have gained a wide popularity in the last decade and they are generally considered to play a fundamental role in coping with the inherent difficulties of the development of large-scale and complex software systems [6]. Software architectures include the early design decisions and embody the overall structure that impacts the quality of the whole system. A common assumption is that architecture design should

support the required software system qualities such as robustness, adaptability, reusability and maintainability [1].

For ensuring the quality factors it is generally agreed that identifying the fundamental abstractions for architecture design is necessary. We maintain that the existing architecture design approaches have several difficulties in deriving the right architectural abstractions. To analyze, evaluate and identify the basic problems we will present a survey of the state-of-the-art architecture design approaches and describe the obstacles of each approach.

The chapter is organized as follows. Section 2 provides a short background on software architectures in which existing definitions including our own definition of software architecture is given. In section 3 a meta-model for software architecture design approaches is presented. This meta-model serves as a basis for identifying the problems in our evaluation of architecture design approaches. In section 4 a classification, analysis and evaluation of the contemporary architectural approaches is presented. Section 5 refers to the related chapters in this volume. Finally, section 6 presents the conclusions and evaluations.

2. NOTION OF ARCHITECTURE

In this section we focus on the meaning of software architecture by analyzing the prevailing definitions as described in section 2.1. In section 2.2 we provide our own definition that we consider as general and which covers the existing definitions.

2.1 Definitions

Software architectures are high-level design representations and facilitate the communication between different stakeholders, enable the effective partitioning and parallel development of the software system, provide a means for directing and evaluation, and finally provide opportunities for reuse [6].

The term architecture is not new and has been used for centuries to denote the physical structure of an artifact [39]. The software engineering community has adopted the term to denote the gross-level structure of software-intensive systems. The importance of structure was already acknowledged early in the history of software engineering. The first software programs were written for numerical calculations using programming languages that supported mathematical expressions and later algorithms and abstract data types. Programs written at that time served mainly one purpose and were relatively simple compared to the current large-scale diverse

software systems. Over time, due to the increasing complexity and the increasing size of the applications, the global structure of the software system became an important issue [30]. Already in 1968, Dijkstra proposed the correct arrangement of the structure of software systems before simply programming [14]. He introduced the notion of layered structure in operating systems, in which related programs were grouped into separate layers, communicating with groups of programs in adjacent layers. Later, Parnas claimed that the selected criteria for the decomposition of a system impact the structure of the programs and several design principles must be followed to provide a good structure [24][25]. Within the software engineering community, there is now an increasing consensus that the structure of software systems is important and several design principles must be followed to provide a good structure [11].

In tandem with the increasing popularity of software architecture design many definitions of software architecture have been introduced over the last decade, though, a consensus on a standard definition is still not established. We think that the reason why so many and various definitions on software architectures exist is because every author approaches a different perspective of the same concept of software architecture and likewise provides a definition from that perspective. Notwithstanding the numerous definitions it appears that the prevailing definitions do not generally conflict with each other and commonly agree that software architecture represents the gross-level structure of the software system consisting of components and relations among them [6][1].

Looking back at the historical developments of architecture design we can conclude that in accordance with many concepts in software engineering the concept of software architecture has also evolved over the years. We observe that this evolution took place at two places. First, existing stable concepts are specialized with new concepts providing a broader interpretation of the concept of software architecture. Second, existing interpretations on software architectures are abstracted and synthesized into new and improved interpretations. Let us explain this considering the development of the definitions in the last decade. The set of existing definitions is large and many other definitions have been collected in various publications such as [34], [27] and [33]. We provide only the definitions that we consider as representative.

[1] Compare this to the parable of "the elephant in the dark", in which four persons are in a dark room feeling different parts of an elephant, and all believing that what they feel is the whole beast.

In the following definition, software architecture represents a high-level structure of a software system. It is in alignment with the earlier concepts of software architecture as described by Dijkstra [14] and Parnas [25].

> *"The logical and physical structure of a system, forged by all the strategic and tactical design decisions applied during development" [8]*

The following definition explicitly considers the interpretation on the elements of software architecture.

> *"We distinguish three different classes of architectural elements: processing elements; data elements; and connection elements. The processing elements are those components that supply the transformation on the data elements; the data elements are those that contain the information that is used and transformed; the connecting elements (which at times may be either processing or data elements, or both) are the glue that holds the different pieces of the architecture together. " [27]*

This definition is a specialization of the previous architecture definitions and represents the functional aspects of the architecture focusing basically on the data-flow in the system. Additional specialization of the structural issues is provided by the following definition.

> *" ...beyond the algorithms and data structures of the computation; designing and specifying the overall system structure emerges as a new kind of problem. Structural issues include gross organization and global control structure; protocols for communication, synchronization, and data access; assignment of functionality to design elements; physical distribution; composition of design elements; scaling and performance; and selection among design alternatives. This is the software architecture level of design. " [16]*

The next definition extends the previous definitions by including design information in the architectural specification.

> *"The structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time." [17]*

Finally, the following definition abstracts from the previous definitions and implies that software architectures have more than one structure and includes the behavior of the components as part of the architecture.

> *"The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them." [6]*

The term component in this definition is used as an abstraction of varying components [6]. This definition may be considered as a sufficiently good representative of the latest abstraction of the concept of software architecture.

2.2 Architecture as a Concept

The understanding on the concept of software architecture is increasing though there are still several ambiguities. Architectures consist of components and relations, but the term components may refer to subsystems, processes, software modules, hardware components or something else. Relations may refer to data flows, control flows, call-relations, part-of relations etc. To provide a consistent and overall definition on architectures, we need to provide an abstract yet a sufficiently precise meaning of the components and relations. For this we provide the following definition of architecture:

> *Architecture is a concept representing a set of abstractions and relations, and constraints among these abstractions.*

In essence this definition considers architecture as a *concept* that is general yet well defined. We think that this definition is general enough to cover the various perspectives on architectures. To clarify this definition and discuss its implications we will provide a closer view on the notion of concept.

A *concept* is usually defined as a (mental) representation of a category of instances [20] and is formed by abstracting knowledge about instances. The process of assigning new instances to a concept is called *categorization* or *classification*. In this context, concepts are also called *categories* or *classes*. There are several theories on concepts and classification addressing the notions of concepts, classes, instances and categories [23][32][26].

In the context of software architectures the architectural concepts are also abstractions of the corresponding domain knowledge. The content of the domain knowledge, however, may vary per architecture design approach.

Concepts are not just arbitrary abstractions or groupings of a set of instances but are defined by a consensus of experts in the corresponding domain. As such concepts are stable and well-defined abstractions with rich semantics. The definition thus enforces that each architecture consists of components that do not only represent arbitrary groupings or categories but are semantically well defined. For a more detailed description of the view on software architecture as a concept, we refer to chapter 3 of [35].

3. META MODEL FOR ARCHITECTURE DESIGN APPROACHES

In this section we provide a meta-model that is an abstraction of various architecture design approaches. We will use this model to analyze and compare current software architecture design approaches.

The meta-model is given in Figure 1. The rounded rectangles represent the concepts, whereas the lines represent the association between these concepts. The diamond symbol represents an association relation between three or four concepts. Let us now describe the concepts individually.

The concept *Client* represents the stakeholder(s) who is/are interested in the development of a software architecture design. A stakeholder may be a customer, end-user, system developer, system maintainer, sales manager etc.

The concept *Domain Knowledge* represents the area of knowledge that is applied in solving a certain problem.

The concept *Requirement Specification* represents the specification that describes the requirements for the architecture to be developed.

The concept *Artifact* represents the artifact descriptions of a certain method. This is, for example, the description of the artifact Class, Operation, Attribute, etc. In general each artifact has a related set of heuristics for identifying the corresponding artifact instances.

The concept *Solution Abstraction* defines the conceptual representation of a (sub)-structure of the architecture.

The concept *Architecture Description* defines a specification of the software architecture.

In Figure 1, there are two quaternary association relations and one ternary association relation.

The quaternary association relation called *Requirements Capturing* defines the association relations between the concepts *Client*, *Domain Knowledge, Requirement Specification* and *Architecture Description*. This

association means that for defining a requirement specification the client, the domain knowledge and the (existing) architecture description can be utilized. The order of processing is not defined by this association and may differ per architecture design approach.

The quaternary association relation called *Extracting Solution Structures* is defined between the concepts *Requirement Specification, Domain Knowledge, Artifact* and *Solution Abstraction*. This describes the structural relations between these concepts to derive a suitable solution abstraction.

The ternary association relation *Architecture Specification* is defined between the concepts *Solution Abstraction*, *Architecture Description* and *Domain Knowledge* and represents the specification of the architecture utilizing these three concepts.

Various architecture design approaches can be described as instantiations of the meta-model in Figure 1. Each approach will differ in the ordering of the processes and the particular content of the concepts.

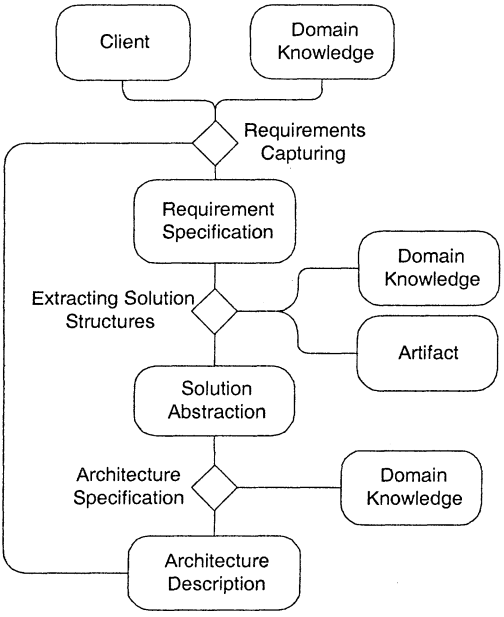


Figure 1: Meta-model for architecture design approaches

In the meta-model, the concept *Domain Knowledge* is used at three different places. Since this concept plays a fundamental role in various architectural design approaches we will now elaborate on this concept.

The term domain has different meanings in different approaches [35]. We distinguish between the following specializations of this concept: *Problem*

Domain Knowledge, *Business Domain Knowledge*, *Solution Domain Knowledge* and *General Knowledge*.

The concept *Problem Domain Knowledge* refers to the knowledge on the problem from a client's perspective. It includes requirement specification documents, interviews with clients, prototypes delivered by clients etc. The concept *Business Domain Knowledge* refers to the knowledge on the problem from a business process perspective. It includes knowledge on the business processes and also customer surveys and market analysis reports. The concept *Solution Domain Knowledge* refers to the knowledge that provides the domain concepts for solving the problem and which is separate from specific requirements and the knowledge on how to produce software systems from this solution domain. This kind of domain knowledge is included in for example textbooks, scientific journals, and manuals. The concept *General Knowledge* refers to the general background and experiences of the software engineer and also may include general rules of thumb. Finally, the concept *System/Product Knowledge* refers to the knowledge about a system, a family of systems or a product.

4. ANALYSIS AND EVALUATION OF ARCHITECTURE DESIGN APPROACHES

A number of approaches have been introduced to identify the architectural design abstractions. We classify these approaches as *artifact-driven*, *use-case-driven* and *domain-driven* architecture design approaches. The criterion for this classification is based on the adopted source for the identification of the key abstractions of architectures. Each approach will be explained as a realization of the meta-model described in Figure 1.

4.1 Artifact-driven Architecture Design

We term artifact-driven architecture design approaches as those approaches that extract the architecture description from the artifact descriptions of the method. Examples of artifact-driven architectural design approaches are the popular object-oriented analysis and design methods such as OMT [29] and OAD [8]. A conceptual model for artifact-driven architectural design is presented in Figure 2. Hereby the labeled arrows represent the process order of the architectural design steps. The concepts *Analysis & Design Models* and *Subsystems* in Figure 2 together represent the concept *Solution Abstraction* of Figure 1. The concept *General Knowledge* represents a specialization of the concept *Domain Knowledge* in Figure 1.

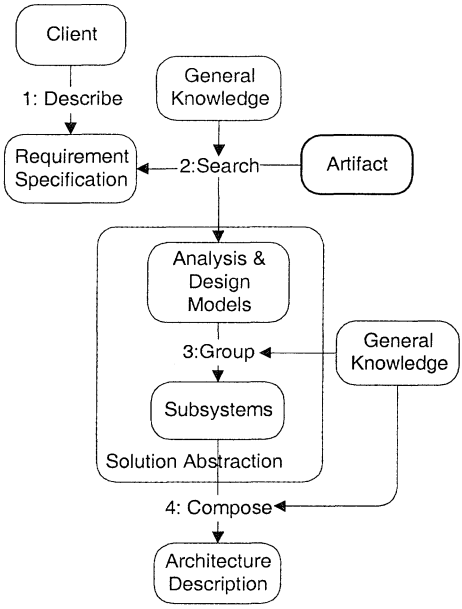


Figure 2: Conceptual model of artifact-driven architectural design

We will explain this model using OMT [29], which can be considered as a suitable representative for this category. In OMT, architecture design is not an explicit phase in the software development process but rather an implicit part of the design phase. The OMT method consists basically of the phases *Analysis, System Design,* and *Object Design.* The arrow *1:Describe* represents the description of the requirement specification. The arrow *2:Search* represents the search for the artifacts such as classes in the requirement specification in the analysis phase. An example of a heuristic rule for identifying tentative class artifacts is the following:

> **IF** an entity in the requirement specification is relevant
> **THEN** select it as a Tentative Class.

The search process is supported by the general knowledge of the software engineer and the heuristic rules of the artifacts that form an important part of the method. The result of the *2:Search* function is a set of artifact instances that is represented by the concept *Analysis &Design Models* in Figure 2.

The method follows with the *System Design* phase that defines the overall architecture for the development of the global structure of a single software system by grouping the artifacts into *subsystems* [29]. In Figure 2, this grouping function is represented by the function *3:Group*. The software

architecture consists of a composition of subsystems, which is defined by the function *4:Compose* in Figure 2. This function is also supported by the concept *General Knowledge.*

4.1.1 Problems

In OMT, the architectural abstractions are derived by grouping a set of classes that are elicited from the requirement specification. We argue that hereby it is difficult to extract the architectural abstractions. We will explain the problems using the example described in [29] on an Automated Teller Machine (ATM) which concerns the design of a banking network. Hereby, bank computers are connected with ATMs from which clients can withdraw money. In addition, banks can create accounts and money can be transferred and/or withdrawn from one account to another. It is further required that the system should have an appropriate record keeping and secure provisions. Concurrent accesses to the same account must be handled correctly.

The problems that we identified with respect to architecture development are as follows:

- *Textual requirements are imprecise, ambiguous or incomplete and are less useful as a source for deriving architectural abstractions*

In OMT, artifacts are searched within the textual requirement specification and grouped into subsystems, which form the architectural components. Textual requirements, however, may be imprecise, ambiguous or incomplete and as such are not suitable as a source for identification of architectural abstractions. In the example, three subsystems are identified: *ATM Stations, Consortium Computer* and *Bank Computers*. These subsystems group the artifacts that were identified from the requirement specification. With respect to the transaction processing, the example only includes one class artifact called *Transaction* since this was the only artifact that could be discovered in the textual requirement specification. Publications on transaction systems, however, show that many concerns such as scheduling, recovery deadlock management etc. are included in designing transaction systems [15][13][7]. Therefore, we would expect additional classes that could not be identified from the requirement specification.

- *Subsystems have poor semantics to serve as architectural components*

In the given example, the component *ATM stations* represent a subsystem, that is, an architectural component. The subsystem concept serves basically as a grouping concept and as such has very poor semantics[2].

[2] In [2] this problem has been termed as subsystem-object distinction.

For the subsystem *ATM stations* it is, for example, not possible to define the architectural properties, architectural constraints with the other subsystems, and the dynamic behavior. This poor semantics of subsystems makes the architecture description less useful as a basis for the subsequent phases of the software development process.

- *Composition of subsystems is not well supported*

Architectural components interact, coordinate, cooperate and are composed with other architectural components. OMT, however, does not provide sufficient support for this process. In the given example, the subsystem *ATM Stations, Consortium Computer* and *Bank Computers* are composed together, though, the rationale for the presented structuring process is performed implicitly. One could provide several possibilities for composing the subsystems. The method, however, lacks rigid guidelines for composing and specifying the interactions between the subsystems.

4.2 Use-Case driven Architecture Design

In the use-case driven architecture design approach, *use cases* are applied as the primary artifacts for deriving the architectural abstractions. A *use case* is defined as a sequence of actions that the system provides for *actors* [21]. Actors represent external roles with which the system must interact. The actors and the use cases together form the *use case model*. The use case model is meant as a model of the system's intended functions and its environment, and serves as a contract between the customer and the developers. The Unified Process [21], for example, applies a use-case driven architecture design approach. The conceptual model for the use-case driven architecture design approach in the Unified Process is given in Figure 3. Hereby, the dashed rounded rectangles represent the concepts of Figure 1. For example the concepts *Informal Specification* and the *Use-Case Model* together form the concept *Requirement Specification* in Figure 1.

The Unified Process consists of *core workflows* that define the static content of the process and describe the process in terms of activities, workers and artifacts. The organization of the process over time is defined by phases. The Unified Process is composed of six core workflows: *Business Modeling*, *Requirements*, *Analysis*, *Design*, *Implementation* and *Test*.

These core workflows result respectively in the following separate models: *business & domain model*, *use-case model*, *analysis model*, *design model*, *implementation model* and *test model*.

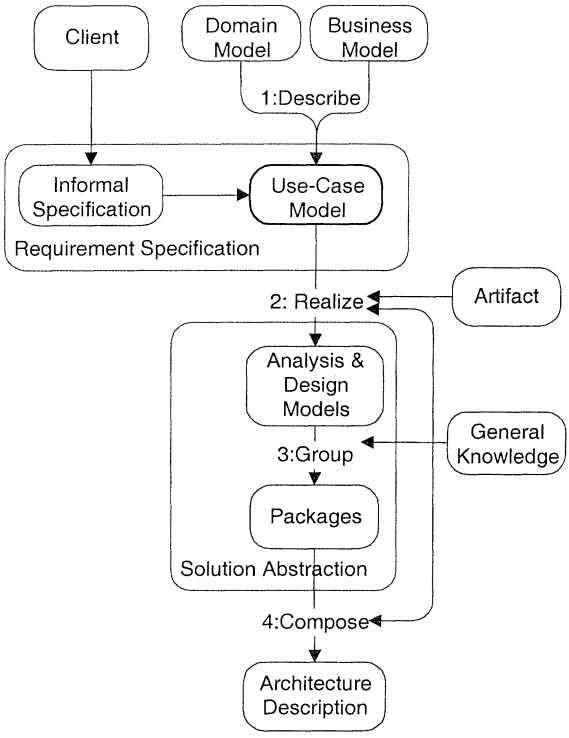


Figure 3: Conceptual model of use-case driven architectural design

In the requirements workflow, the client's requirements are captured as use cases which results in the use-case model. This process is defined by the function *1:Describe* in Figure 3. Together with the informal requirement specification, the use case model forms the requirement specification. The development of the use case model is supported by the concepts *Informal Specification*, *Domain Model* and *Business Model* that are required to define the system's context. The *Informal Specification* represents the textual requirement specification. The *Business Model* describes the business processes of an organization. The *Domain Model* describes the most important classes within the context of the domain. From the use case model the architecturally significant use cases are selected and *use-case realizations* are created as it is described by the function *2:Realize*. Use case realizations determine how the system internally performs the tasks in terms of collaborating objects and as such help to identify the artifacts such as classes. The use-case realizations are supported by the knowledge on the corresponding artifacts and the general knowledge. This is represented by the arrows directed from the concepts *Artifact* and *General Knowledge* respectively, to the function *2:Realize*. The output of this function is the

concept *Analysis & Design Models*, which represents the identified artifacts after use-case realizations.

The analysis and design models are then grouped into *packages*, which is represented by the function *3:Group*. The function *4:Compose* represents the definition of interfaces between these packages resulting in the concept *Architecture Description*. Both functions are supported by the concept *General Knowledge*.

4.2.1 Problems

In the Unified Process, first the business model and the domain model are developed for understanding the context. Use case models are then basically derived from the informal specification, the business model and the domain model. The architectural abstractions are derived from realizations of selected use cases from the use case models.

We think that this approach has to cope with several problems in identifying the architectural abstractions. We will motivate our statements using the example described in [21, pp. 113] that concerns the design of an electronic banking system in which the internet will be used for trading of goods and services and likewise include sending orders, invoices, and payments between sellers and buyers. The problems that we encountered are listed as follows:

- *Leveraging detail of domain model and business model is difficult*

The business model and domain models are defined before the use case model. The question raises then how to leverage the detail of these models. Before use cases are known it is very difficult to answer this question since use cases actually define what needs to be developed. In [21, pp. 120] a domain model is given for an electronic banking system example. Domain models are derived from domain experts and informal requirement specifications. The resulting domain model includes four classes: *Order*, *Invoice*, *Item* and *Account*. The question here is whether these are the only important classes in electronic banking systems. Should we consider also the classes such as *Buyer* and *Seller*? The approach does not provide sufficient means for defining the right detail of the domain and business models[3].

[3] Use cases focus on the functionality for each user of the system rather than just a set of functions that might be good to have. In that sense, use cases form a practical aid for leveraging the requirements. They are however less practical for leveraging the domain and business models.

- *Selecting architecturally relevant use-cases is not systematically supported*

For the architecture description, 'architecturally relevant' use cases are selected. The decision on which use cases are relevant lacks objective criteria and is merely dependent on some heuristics and the evaluation of the software engineer. For example, in the given banking system example, the use case *Withdraw Money* has been implicitly selected as architecturally relevant and other use cases such as *Deposit Money* and *Transfer between Accounts* have been left out.

- *Use-cases do not provide a solid basis for architectural abstractions*

After the relevant use cases have been selected they are *realized*, which means that analysis and design classes are identified from the use cases. Use-case realizations are supported by the heuristic rules of the artifacts, such as classes, and the general knowledge of the software engineer. This is similar to the artifact-driven approach in which artifacts are discovered in the textual requirements. Although use cases are practical for understanding and representing the requirements, we maintain that they do not provide a solid basis for deriving architectural design abstractions. Use cases focus on the problem domain and the external behavior of the system. During use case realizations, transparent or hidden abstractions that are present in the solution domain and the internal system may be difficult to identify. Thus even if all the relevant use cases have been identified it may still be difficult to identify the architectural abstractions from the use case model. In the given banking system example, the use case-realization of *Withdraw Money* results in the identification of the four analysis classes *Dispenser*, *Cashier Interface*, *Withdrawal* and *Account* [21, pp. 44]. The question here is whether these are all the classes that are concerned with withdrawal. For example, should we also consider classes such as *Card* and *Card Check*? The transparent classes cannot be identified easily if they have not been described in the use case descriptions.

- *Package construct has poor semantics to serve as an architectural component*

The analysis and design models are grouped into package constructs. Packages are, similar to subsystems in the artifact-driven approach, basically grouping mechanisms and as such have poor semantics. The grouping of analysis and design classes into packages and the composition of the packages into the final architecture are also not well supported and basically depend on the general knowledge of the software engineer. This may again

lead to ill-defined boundaries of the architectural abstractions and their interactions.

- *Applied heuristics are implicitly based on the structural paradigm and hinder to identify and define the fundamental abstractions of current large scale and diverse applications.*

A close look at the Unified Process results in the interesting observation that the heuristics that are applied to identify the abstractions, which are needed to define the architectural components, are essentially based on the traditional structural paradigm of software development [38]. In this paradigm, data is processed by a set of functions resulting in some output data. The basic abstractions that define the architectural components in the Unified Process are the classes and packages. Classes are derived from the use case model by searching for entities that are needed for interfacing, control and information. Packages are derived from the problem domain and use cases that support specific business processes, require specific actors or use cases that are related via generalizations and extends-relations. Both in class identification and package identification, actually functional entities are searched that get some input data, process these and provide some output data. This bias from the early period of software engineering, which largely dealt with defining systems for numerical applications, is not suitable anymore for identifying and defining the architectural abstractions of current large-scale, diverse systems. According to the definition of architecture in section 2.2, the components of architecture represent the fundamental abstractions of the domain. From this perspective, the architectural components may also correspond to non-functional abstractions.

4.3 Domain-driven Architecture Design

Domain-driven architecture design approaches derive the architectural design abstractions from domain models. The conceptual model for this domain-driven approach is presented in Figure 4.

Domain models are developed through a domain analysis phase represented by the function *2:Domain Analysis*. Domain analysis can be defined as the process of identifying, capturing and organizing domain knowledge about the problem domain with the purpose of making it reusable when creating new systems [28]. The function *2:Domain Analysis* takes as input the concepts *Requirement Specification* and *Domain Knowledge* and results in the concept *Domain Model.* Note that both the concepts *Solution Domain Knowledge* and *Domain Model* in Figure 4 represent the concept *Domain Knowledge* in the meta-model of Figure 1.

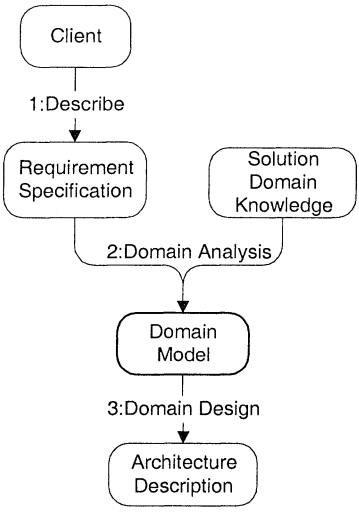


Figure 4: Conceptual model for Domain-Driven Architecture Design

The domain model may be represented using different representation forms such as classes, entity-relation diagrams, frames, semantics networks, and rules. Several *domain analysis* methods have been published, e.g. [18], [22], [28], [31] and [12]. Two surveys of various domain analysis methods can be found in [3] and [40]. In [12] a more recent and extensive up-to-date overview of domain engineering methods is provided.

In this chapter we are mainly interested in the approaches that use the domain model to derive architectural abstractions. In Figure 4, this is represented by the function *3:Domain Design.* In the following we will consider two domain-driven approaches that derive the architectural design abstractions from domain models.

4.3.1 Product-line Architecture Design

In the product-line architecture design approach, an architecture is developed for a *software product-line* that is defined as a group of software-intensive products sharing a common, managed set of features that satisfy the needs of a selected market or mission area [10]. A *software product line architecture* is an abstraction of the architecture of a related set of products. The product-line architecture design approach focuses primarily on the reuse within an organization and consists basically of *the core asset development* and *the product development.* The core asset base often includes the architecture, reusable software components, requirements, documentation and specification, performance models, schedules, budgets, and test plans

and cases [4], [5], [10]. The core asset base is used to generate or integrate products from a product line.

The conceptual model for product-line architecture design is given in Figure 5. The function *1:Domain Engineering* represents the core asset base development. The function *2:Application Engineering* represents the product development from the core asset base.

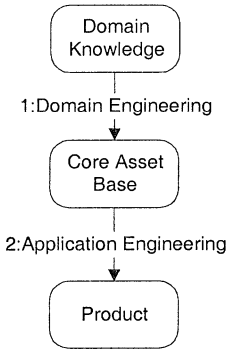


Figure 5: A conceptual model for a Product-Line Architecture Design

Note that various software architecture design approaches can be applied to provide a product-line architecture design. In the following section we will describe an approach that follows the conceptual model for product-line architecture design in Figure 5.

4.3.2 Domain Specific Software Architecture Design

The *domain-specific software architecture* (DSSA) [19][37] may be considered as multi-system scope architecture, that is, it derives an architectural description for a family of systems rather than a single-system. The conceptual model of this approach is presented in Figure 6.

The basic artifacts of a DSSA approach are the *domain model*, *reference requirements* and the *reference architecture*. The DSSA approach starts with a domain analysis phase on a set of applications with common problems or functions. The analysis is based on *scenarios* from which functional requirements, data flow and control flow information is derived. The *domain model* includes scenarios, domain dictionary, context (block) diagrams, ER diagrams, data flow models, state transition diagrams and object models.

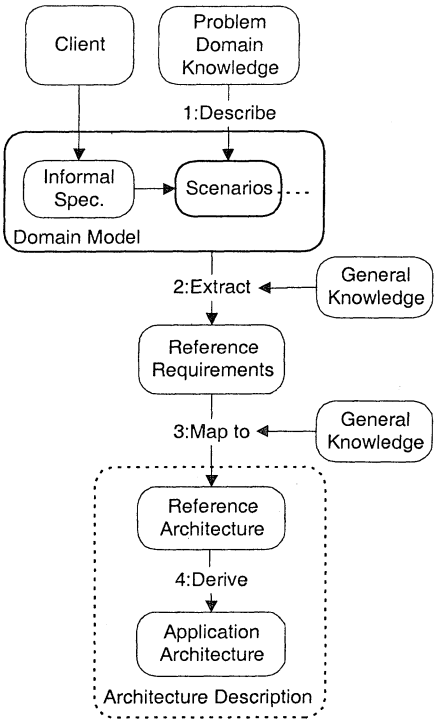


Figure 6: Conceptual model for Domain Specific Software Architecture (DSSA) approach

In addition to the domain model, *reference requirements* are defined that include functional requirements, non-functional requirements, design requirements and implementation requirements and focus on the solution space. The domain model and the reference requirements are used to derive the *reference architecture*. The DSSA process makes an explicit distinction between a *reference architecture* and an *application architecture*. A reference architecture is defined as the architecture for a family of application systems, whereas an application architecture is defined as the architecture for a single system. The application architecture is instantiated or refined from the reference architecture. The process of instantiating/refining and/or extending a reference architecture is called *application engineering*.

4.3.3 Problems

Since the term domain is interpreted differently there are various domain-driven architecture design approaches. We list the problems for problem domain analysis and solution domain analysis.

- *Problem domain analysis is less effective in deriving architectural abstractions*

Several domain-driven architecture approaches interpret the domain as a problem domain. The DSSA approach, for example, starts from an informal problem statement and derives the architectural abstractions from the domain model that is based on scenarios. Like use cases, scenarios focus on the problem domain and the external behavior of the system. We think that approaches that derive abstractions from the problem domain, such as the DSSA approach, are less effective in deriving the right architectural abstractions. Let us explain this using the example in [36] in which an architecture for a theater ticket sales application is constructed using the DSSA approach. In this example a number of scenarios such as *Ticket Purchase*, *Ticket Return, Ticket Exchange*, *Ticket Sales Analysis*, and *Theater Configuration* are described and accordingly a domain model is defined based on these scenarios. The question hereby is whether the given scenarios fully describe the system and as such result in the right scoping of the domain model. Are all the important abstractions identified? Do there exist redundant abstractions? How can this be evaluated? Within this approach and other approaches that derive the abstractions from the problem domain these questions remain rather unanswered.

- *Solution domain analysis is not sufficient*

Although solution domain analysis provides the potential for modeling the whole domain that is necessary to derive the architecture, it is not sufficient to drive the architecture design process. This is due to two reasons. First, solution domain analysis is not defined for software architecture design per se, but rather for systematic reuse of assets for activities in, for example, software development. Since the area on which solution domain analysis is performed may be very wide, it may easily result in a domain model that is too large and includes abstractions that are not necessary for the corresponding software architecture construction. The large size of the domain model may hinder the search for the architectural abstractions. The second problem is that the solution domain may not be sufficiently cohesive and stable to provide a solid basis for architectural design. Concepts in the corresponding solution domain may not have reached a consensus yet and still be under development. Obviously, one cannot expect to provide an architecture design solution that is better than the solution provided by the solution domain itself. Therefore, a thorough solution domain analysis may in this case also not be sufficient to provide stable abstractions.

5. OVERVIEW OF THE ARCHITECTURE DESIGN APPROACHES IN THIS BOOK

The chapters in part 2 of this book are devoted to the architectural issues in software development. Chapters 3, 4 and 5 present architecture design methods. Chapter 6 proposes a set of basic abstractions to design various kinds of message-based architectures. Chapters 7 and 8 describe means to refine architectures into object-oriented software systems. In the following, we give a brief summary of these chapters from the perspective of architecture design.

Chapter 3 emphasizes the importance of the so-called non-functional properties in architecture design. By using a car navigation system design example, the chapter illustrates how the functional and non-functional requirements can be considered in a uniform manner. The functional design of the architecture is use-case driven, and therefore, confirms to the design model shown in Figure 3. The non-functional part of the design is, however, solution-domain driven and can be considered as an instantiation of the design model shown in Figure 4. For example, deadline and schedulability solution-domain techniques are used to analyze and design the non-functional characteristics of the architecture.

In chapter 4, by the help of two industrial design problems, a product-line architecture design method is presented. This is a domain-driven architecture design approach, which confirms the design models shown in figure 4 and figure 5. The architectural abstractions are derived by using both use-cases and solution domain abstractions.

In chapter 5, a synthesis-based architecture design method is presented. In this method, the functional requirements, which may be derived from the use cases, are first expressed in terms of technical problems. These problems are then synthesized towards solutions by systematically applying solution domain knowledge. This approach can be considered as a specialization of the domain-driven architecture design method shown in Figure 4.

In chapter 6, first various message-oriented interaction styles are analyzed. A set of basic abstractions is derived from the solution domains such message-oriented interaction models, delivery semantics and reliability concerns. The expressiveness of these abstractions are motivated by a number of examples.

Chapter 7 proposes a logic meta-programming language to capture and preserve the architectural constraints along the refinement process.

Chapter 8 proposes a technique called Design Algebra to analyze and refine various architecture implementation alternatives by using quality factors such as adaptability and performance.

6. CONCLUSION

In this chapter we have defined architecture as a set of abstractions and relations that together form a concept. Further, a meta-model that is an abstraction of software architecture design approaches is provided. We have used this model to analyze, compare and evaluate architecture design approaches. These approaches have been classified as *artifact-driven*, *use-case-driven* and *domain-driven* architecture design approaches. The criterion for this classification is based on the adopted source for the identification of the key abstractions of architectures. In the *artifact-driven* approaches the architectural abstractions are represented by groupings of artifacts that are elicited from the requirement specification. *Use-case driven* approaches derive the architectural abstractions from use case models that represent the system's intended functions. *Domain-driven* architecture design approaches derive the architectural abstractions from the domain models. For each approach, we have described the corresponding problems and motivated why these sources are not optimal in identifying the architectural abstractions. We can abstract the problems basically as follows:

1. Difficulties in Planning the Architectural Design Phase

Planning the architecture design phase in the software development process is a dilemma[4]. In general, architectures are identified before or after the analysis and design phases. Defining the architecture can be done more accurately after the analysis and design models have been determined because these impact the boundaries of the architecture. This may lead, however, to an unmanageable project because the architectural perspective in the software development process will be largely missing. On the other hand, planning the architecture design phase before the analysis and design phases may also be problematic since the architecture may not have optimal boundaries due to insufficient knowledge on the analysis and design models[5].

In artifact-driven architecture design approaches the architecture phase follows after the analysis and design phases and as such the project may become unmanageable. In the domain-driven architecture design approaches the architecture design phase follows a domain engineering phase in which first a domain model is defined from which consequently architectural abstractions are extracted. Hereby the architecture definition may be

[4] In [2] this problem has been denoted as the *early decomposition* problem.

[5] An analogy of this problem is writing an introduction to a book. To organize and manage the work on the different chapters it is required to provide a structure of the chapters in advance. However, the final structure of the introduction can be usually only defined after the chapters have been written and the complete information on the structure is available.

unmanageable if the domain model is too large. In the use-case driven architecture design approach the architecture definition phase is part of the analysis and design phase and the architecture is developed in an iterative way. This does not completely solve the dilemma since the iterating process is mainly controlled by the intuition of the software engineer.

2. *Client requirements are not a solid basis for architectural abstractions*

The client requirements on the software-intensive system that needs to be developed is different from the architectural perspective. The client requirements provide a problem perspective of the system whereas the architecture is aimed to provide a solution perspective that can be used to realize the system. Due to the large gap between the two perspectives the architectural abstractions may not be directly obvious from the client requirements. Moreover, the requirements themselves may be described inaccurately and may be either under-specified or over-specified. Therefore, sometimes it is also not preferable to adopt the client requirements.

This problem is apparent in all the approaches that we analyzed. In the artifact-driven approach the client requirements are directly used as a source for identifying the architectural abstractions. The use-case driven approach attempts to model the requirements also from a client perspective by utilizing use case models. In the domain-driven approaches, such as the domain specific software architecture design approach (DSSA), informal specifications are used to support the development of scenarios that are utilized to develop domain models.

3. *Leveraging the domain model is difficult*

The domain-driven and the use case approaches apply domain models for the construction of software architecture. Uncontrolled domain engineering may result in domain models that lack the right detail of abstraction to be of practical use. The one extreme of the problem is that the domain model is too large and includes redundant abstractions; the other extreme is that it is too small and misses the fundamental abstractions. Domain models may also include both redundant abstractions and still miss some other fundamental abstractions. It may be very difficult to leverage the detail of the domain model.

This problem is apparent in domain-driven and the use-case driven approaches. In the domain-driven approaches that derive domain models from problem domains, such as the DSSA approach, leveraging the domain model is difficult because it is based on scenarios that focus on the system from a problem perspective rather than a solution perspective. In the use-case driven architecture design approach, leveraging the domain model and

business model is difficult since it is performed before use-case modeling and it is actually not exactly known what is desired.

4. *Architectural abstractions have poor semantics*

A software architecture is composed of architectural components and architectural relations among them. Often architectural components are similar to groupings of artifacts, which are named as subsystems, packages etc. These constructs do not have sufficiently rich semantics to serve as architectural components. Architectural abstractions should be more than grouping mechanisms and the nature of the components and their relations, and the architectural properties, the behavior of the system should be described as well [9]. Because of the lack of semantics of architectural components it is very hard to understand the architectural perspective and make the transition to the subsequent analysis and design models.

5. *Composing architectural abstractions is weakly supported*

Architectural components interact, coordinate, cooperate and are composed with other architectural components. The architecture design approaches that we evaluated do not provide, however, explicit support for composing architectural abstractions.

ACKNOWLEDGEMENTS

This research has been supported and funded by various organizations including Siemens-Nixdorf Software Center, the Dutch Ministry of Economical affairs under the SENTER program, the Dutch Organization for Scientific Research (NWO, 'Inconsistency management in the requirements analysis phase' project), the AMIDST project, and by the IST Project 1999-14191 EASYCOMP.

7. REFERENCES

1. G. Abowd, L. Bass, R. Kazman and M. Webb. *SAAM: A Method for Analyzing the Properties of Software Architectures*. In: Proceedings of the 16th International Conference on Software Engineering, CA: IEEE Computer Society Press, pp. 81-90, May, 1994.
2. M. Akşit and L. Bergmans. *Obstacles in Object-Oriented Software Development*. In Proceedings OOPSLA '92, ACM SIGPPLAN Notices, Vol. 27, No. 10, pp. 341-358, October 1992.
3. G. Arrango. *Domain Analysis Methods*. In: Software Reusability, Schäfer, R. Prieto-Díaz, and M. Matsumoto (Eds.), Ellis Horwood, New York, New York, pp. 17-49, 1994.

4. L. Bass, P. Clements, S. Cohen, L. Northrop, and J. Withey. *Product Line Practice Workshop Report*, Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1997.
5. L. Bass, P. Clements, G. Chastek, S. Cohen, L. Northrop and J. Withey. *2nd Product Line Practice Workshop Report*, Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1997.
6. L. Bass, P. Clements and R. Kazman. *Software Architecture in Practice*, Addison-Wesley 1998.
7. P.A. Bernstein and E. Newcomer. *Principles of Transaction Processing*, Morgan Kaufman Publishers, 1997.
8. G. Booch. *Object-Oriented Design with Applications*, The Benjamin/Cummings Publishing Company, Inc, 1991.
9. P. Clements. *A Survey of Architectural Description Languages*. In: Proceedings of the 8th International Workshop on Software Specification and Design, Paderborn, Germany, March, 1996.
10. P. Clements and L.M. Northrop. *Software Architecture: An Executive Overview*, Technical Report, CMU/SEI-96-TR-003, Carnegie Mellon University, 1996.
11. P. Clements, D. Parnas and D. Weiss. *The Modular Structure of Complex Systems*. IEEE Transactions on Software Engineering, Vol. 11, No. 1, pp. 259-266, 1985.
12. K. Czarnecki & U.W. Eisenecker. *Generative Programming*: Methods, Tools, and Applications. Addison-Wesley, 2000.
13. C.J. Date. *An Introduction to Database Systems*, Vol. 3, Addison Wesley, 1990.
14. E.W. Dijkstra. *The Structure of the 'T.H.E.' Mulitprogramming System.* Communications of the ACM, Vol. 18, No. 8, pp. 453-457, 1968.
15. A.K. Elmagarmid (ed.) *Transaction Management in Database Systems*, Morgan Kaufmann Publishers, 1991.
16. D. Garlan and M. Shaw. *An Introduction to Software Architecture*. Advances in: Software Engineering and Knowledge Engineering. Vol 1. River Edge, NJ: World Scientific Publishing Company, 1993.
17. D. Garlan, R. Allen and J. Ockerbloom. *Architectural Mismatch: Why It's Hard to Build Systems Out of Existing Parts*. In: Proceedings of the 17th International Conference on Software Engineering. Seattle, WA, April 23-30, 1995. New York: Association for Computing Machinery, pp. 170-185, 1995.
18. H. Gomaa. *An Object-Oriented Domain Analysis and Modeling Method for Software Reuse*. In: Proceedings of the Hawaii International Conference on System Sciences, Hawaii, January, 1992.
19. F. Hayes-Roth. *Architecture-Based Acquisition and Development of Software: Guidelines and Recommendations from the ARPA Domain-Specific Software Architecture (DSSA) Program*. Version 1.01, Technical Report, Teknowledge Federal Systems, 1994.
20. R.W. Howard. *Concepts and Schemata*: *An Introduction*, Cassel Education, 1987.
21. I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.
22. K. Kang, S. Cohen, J. Hess, W. Nowak and S. Peterson. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report, CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, November 1990.
23. G. Lakoff. *Women, Fire, and Dangerous Things: What Categories Reveal about the Mind*, The University of Chicago Press, 1987.

24. D. Parnas. *On the Criteria for Decomposing Systems into Modules.* Communications of the ACM, Vol. 15, No. 12, pp. 1053-1058, 1972.
25. D. Parnas. *On the Design and Development of Program Families.* IEEE Transactions on Software Engineering SE-2, 1: 1-9, 1976.
26. J. Parsons and Y. Wand. *Choosing Classes in Conceptual Modeling*, Communications of the ACM, Vol 40. No. 6., pp. 63-69, 1997.
27. D.E. Perry and A.L. Wolf. *Foundations for the Study of Software Architecture.* Software Engineering Notes, ACM SIGSOFT 17, No. 4, pp. 40-52, October 1992.
28. R. Prieto-Diaz and G. Arrango (Eds.). *Domain Analysis and Software Systems Modeling.* IEEE Computer Society Press, Los Alamitos, California, 1991.
29. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy and W. Lorensen. *Object-Oriented Modeling and Design*, Prentice Hall, 1991.
30. M. Shaw and D. Garlan. *Software Architectures: Perspectives on an Emerging Discipline*, Englewood Cliffs, NJ: Prentice-Hall, 1996.
31. M. Simos, D. Creps, C. Klinger, L. Levine and D. Allemang. *Organization Domain Modeling (ODM) Guidebook*, Version 2.0. Informal Technical Report for STARS, STARS-VC-A025/001/00, June 14, http://www.synquiry.com, 1996.
32. E.E. Smith and D.L. Medin. *Categories and Concepts*, Harvard University Press, London, 1981.
33. Software Engineering Institute, Carnegie Mellon university, Web-site: http://www.sei.cmu.edu/architecture/, 2000.
34. D. Soni, R. Nord and C. Hofmeister. *Software Architecture in Industrial Applications.* 196-210. Proceedings of the 17th International ACM Conference on Software Engineering, Seattle, WA, 1995.
35. B. Tekinerdoğan. *Synthesis-Based Software Architecture Design*, PhD Thesis, Dept. Of Computer Science, University of Twente, March 23, 2000.
36. W. Tracz. *DSSA (Domain-Specific Software Architecture) Pedagogical Example.* In: ACM SIGSOFT Software Engineering Notes, Vol. 20, No. 4, July 1995.
37. W. Tracz and L. Coglianese. *DSSA Engineering Process Guidelines. Technical Report.* ADAGE-IBM-9202, IBM Federal Systems Company, December, 1992.
38. E. Yourdon. *Modern Structured Analysis.* Yourdon Press, 2000.
39. Webster on-line Dictionary, http://www.m-w.com/cgi-bin/dictionary, 2000.
40. S. Wartik and R. Prieto-Díaz. *Criteria for Comparing Domain Analysis Approaches.* In: International Journal of Software Engineering and Knowledge Engineering, Vol. 2, No. 3, pp. 403-431, September 1992.

Chapter 2

GUIDELINESS FOR IDENTIFYING OBSTACLES WHEN COMPOSING DISTRIBUTED SYSTEMS FROM COMPONENTS

Mehmet Akşit and Lodewijk Bergmans
TRESE Group, Department of Computer Science, University of Twente, postbox 217, 7500 AE, Enschede, The Netherlands. email: {aksit, bergmans}@cs.utwente.nl, www: http://trese.cs.utwente.nl

Keywords: Distributed systems, aspects of computer science sub-domains, obstacles in component composition, research in component composition

Abstract: Component-oriented programming enables software engineers to implement complex applications from a set of pre-defined components. Although this technique has several advantages, experiences show that effective composition of components remains a difficult task. This is especially true if the software system is physically distributed. This chapter provides a set of guidelines to identify the obstacles that software engineers may experience while designing distributed systems using the current component technology. To this aim, the computer science domain is divided into several sub-domains, and each sub-domain is described using its important aspects. Further, each aspect is analyzed with respect to the current component technology. This analysis helps software engineers to identify the possible obstacles for each aspect of a sub-domain. The chapter concludes with references to the relevant research activities that are presented in this book.

1. DEFINITIONS

1.1 Middleware Systems

To identify the obstacles related to composing components, we will refer to the distributed system architecture shown in Figure 1. From an abstract view, a distributed system can be divided into two layers: software applications and the middleware. We assume that software applications provide services to an environment, which is considered external to the software system. Application services can be considered specific, evolving and diverse. The middleware abstracts the underlying computing and networking technology and provides services that are required by most of applications.

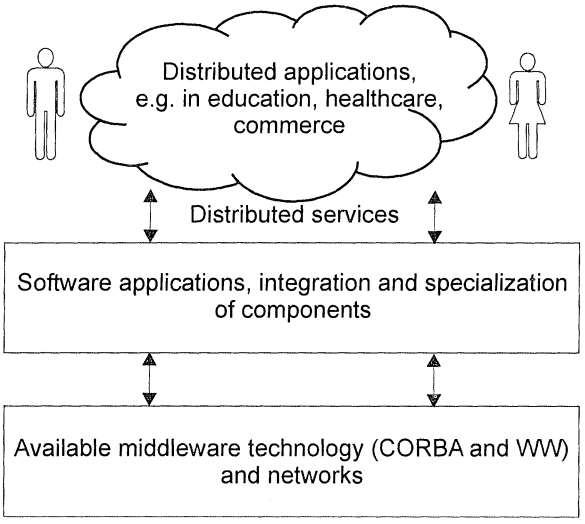


Figure 1: Reference middleware architecture

The main motivation in adopting a middleware is economics. Instead of repeatedly implementing services that are required by most distributed applications each time an application is developed, it is more economic to provide these generic services by the middleware system. For example, almost every distributed application requires a name server and remote invocation mechanism. Many distributed applications require transaction and security services. To determine the services required for a particular middleware system, it is necessary to enumerate the services that are

required by most applications likely to be installed on the middleware. Accordingly, the middleware system should provide these services.

1.2 The Obstacles Caused by Complexity and Evolution of Applications

Unfortunately, complexity and evolution of applications make designing distributed applications a difficult task.

Complexity may hinder a proper decomposition of the software systems into autonomous modules. Moreover, some aspects of the applications may not be expressed sufficiently by the adopted design and/or language models. We term the first problem as the *decomposition* problem and the latter as the *lack of expression* power problem.

We observe the following three affects of evolution: Firstly, the application domain of the middleware technology grows steadily. For instance, nomadic and agent-based computing are some examples of new developments in the area of distributed systems. These new applications generally require extensions to the current middleware services.

Secondly, demands for extensions to existing services require modifications to middleware. For example, most business applications today require support for flexible transactions, whereas current middleware systems generally provide strict serialization and recovery.

Finally, it is becoming more and more common that middleware systems offer services of different quality. These, the so-called *quality of services* can be defined in terms of various parameters such as performance, reliability, and security. The users of a system can be allowed to select the required quality of a service with respect to a certain cost.

The middleware designers may attempt to solve the above-mentioned problems by implementing a dedicated service for each particular service demand. Since demands are evolving and diverse, this would require continuous implementations of many services, which is unfeasible. Instead of implementing a dedicated set of application services, it may be more feasible to compose application services from simpler components that correspond to the fundamental aspects of the application being designed. Obviously component composition here plays a major role. The difficulties that the designers may experience in composing components are termed as the *composition* problem.

1.3 Identification of Obstacles Using Domain Analysis

In this chapter we adopt domain analysis techniques to identify the obstacles in designing distributed systems[1]. Figure 2 illustrates the domain analysis process adopted in this chapter. The first stage is to formulate the technical problems in the requirement specification. Second, these problems are used to select the corresponding solution domains. Third, if the solution domains are found, then the aspects of these domains are determined. Fourth, the expected obstacles are identified for each aspect. Finally, the obstacles are brought into the context of the application.

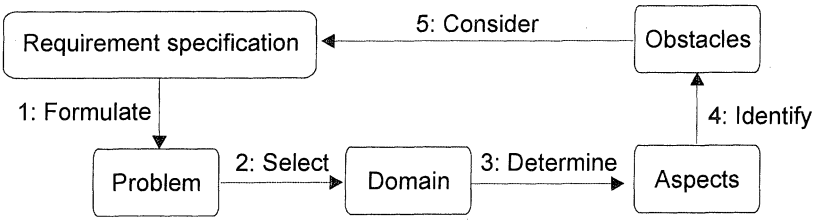


Figure 2: The domain analysis process

In general, the domain of a software system can be intuitively divided into three, possibly overlapping categories:

- Application domain;
- Mathematical domain;
- Computer science domain.

The application domain corresponds to the specific concepts of the application being developed. Assume for example that we want to design a distributed container transport system, which has two main tasks: allocation of containers to the vessels, and creating route plans for the vessels to transport containers among seaports. The application domain here deals with the specific application features of a container transport system such as modeling containers, vessels and seaports.

The mathematical domain deals with the concepts studied within the area of mathematical sciences. In the container transport system example, allocation of containers to vessels and routing vessels among seaports can be seen as mathematical optimization problems.

Since we are interested in software realization techniques, the topics studied in computer science must be considered as well. In the container

[1] For a more detailed description of domains, the reader may refer to chapter 1 of this book.

transport system example, a possible computer science domain is distributed systems since the container transport system is expected to run on multiple sites, such as at vessels and seaports.

Application domains are generally very diverse and therefore it is not possible to reason about the domain of an application without precisely specifying it. The mathematical domain is considered out of the scope of this chapter. In the following sections, the main focus will be on the computer science domain.

The remaining sections of this chapter are organized as follows. The assumptions in analyzing the computer science domain are presented in the following section. Further, aspects general to all sub-domains are introduced and the expected obstacles per aspect are identified. The obstacles are printed as underlined and in Italics. In section 3, the computer science domain is divided into several sub-domains. Each sub-domain is described by using its important aspects and for each aspect the expected obstacles are described. Finally, section 4 gives conclusions. Appendix summarizes the identified obstacles and refers to the relevant sections of this chapter.

2. IDENTIFYING THE OBSTACLES BY ANALYZING THE COMPUTER SCIENCE DOMAIN

2.1 Assumptions

To identify the obstacles of using components from the computer science perspective, we make the following assumptions:

- The computer science domain can be decomposed into sub-domains such as application generators, concurrent processing, constraint systems, control systems, distributed systems and real-time systems.
- The level of detail of each sub-domain can be quite different. For example, the domain of concurrency and synchronization can be considered to be more basic than application generator design.
- Each sub-domain is specified in terms of its aspects. The aspects of a domain are the important features that distinguish that domain from others.
- The intention is to provide intuitive and somewhat historical classification of the computer science domain. To further abstract the aspects of sub-domains may result in too abstract models (such as

lambda calculus), which we consider less suitable for the purpose of this chapter.

- While we are listing the aspects of a sub-domain, we prefer to include aspects only specific to that sub-domain, although in practice, very few pure domain-specific applications are developed. For example, although almost all distributed systems provide some degree of concurrency, we will treat concurrency separate from the distribution aspects.
- While defining the computer science sub-domains, we do not intent to analyze application specific issues but rather we want to know what kind of computation models are needed to support applications in that sub-domain. For example, when we talk about the security aspects in distributed systems and try to identify the problems related to these, we do not intent to assess the basic security enforcement techniques such as encryption, decryption and digital signatures. But rather, we would like to assess the applicability of the component model to support secure distributed systems.
- Some aspects are applicable to all sub-domains.

2.2 The Common Aspects of all Domains

The aspects described in subsections 2.2.1 to 2.2.7 are general and maybe used in supporting all kinds of distributed applications and middleware systems.

2.2.1 Components, Objects and Classes

Components are defined as autonomous software modules with well-defined interfaces. In the literature the term component generally refers to the abstractions provided by the enabling technology such as CORBA, DCOM, OLE, ActiveX and JavaBeans [17]. Objects are instantiations of classes, provide encapsulation and are characterized by well-defined interfaces. Component and objects are related to each other in that any composite and autonomous object structure, in principle, can be considered as a component[2]. The component concept is general and suitable for constructing distributed applications and middleware systems.

[2] We will use the term component for components provided by the enabling technology, and for autonomous objects specified by an object-oriented language.

2.2.2 Interface Declarations and Type Checking

One of the fundamental characteristics of components is the specification of the functional interface. For example, CORBA introduces the Interface Definition Language (IDL) for specifying the interfaces of CORBA components. Strongly typed object-oriented languages provide type-checking mechanism based on the object (class) interface concept. Further, class hierarchies can be used to define sub-type hierarchies.

Although typing the interface of components is useful for early detecting the interaction errors, problems can be experienced when all the meaningful combinations of components have to be declared as a separate type module. In this case, the designers may be forced to declare a large number of type modules.

Assume for example that a car game is to be constructed from components. The Abstract Factory pattern [10] is used for creating various car models in a flexible way. In this pattern, the factory component provides a set of operations to create consistent car components so that a car model can be safely composed from these components. Here, the factory component can be considered as the type of a consistent car model. As a result, for each version of a car model a different factory is necessary. Since a car model may have many different versions, it is necessary to define many factories. This problem is termed as the *excessive type declarations* problem.

2.2.3 Encapsulation and Multiple Interfaces

Encapsulation is an essential property of components and supported by all component models. Depending on the context, distributed applications may require some means to control the visibility of operations of a component.

Assume for example that a mail component migrates over a network passing through different layers. This generally requires changing the interface of the mail component based on its context. For example, the mail sender must have an access to the interface for creating the mail attributes such as the content. The mail-system layer must have an access to the interface of the mail component for approving and delivering the mail. It is not desirable, for example that the mail sender approves the mail and the mail system reads the mail content. This means that every mail component must be able to check the identity of the sender of a request, for example, before returning the mail content. The current component models, however, may have difficulties in detecting the *sender* of a request. Moreover, if the view checking is realized in the implementation of an operation, then the view checking and operation semantics become not separable. This makes a

separate extension of the view checking mechanism and/or operation semantics difficult. Note that introducing a different implementation component for every interface cannot provide the necessary solution. This is because there must be one component, with a single identity, and a single state, which behaves differently according to the way it is being interfaced. This problem is termed as the *multiple-views* problem [2][5][7].

2.2.4 Message Passing

Message passing is the basic mechanism to express coordination among components. Applications may demand flexible message passing semantics from the middleware. Assume for example that an office workflow system has to be designed for supporting various office activities such as reporting, planning, internal request handling, agenda management and electronic meeting. Several of these activities must be executed concurrently. Further, electronic meetings may require stream-based communication.

In current component models, reuse and extensibility issues focus on extending and redefining the features of components. However, the concept of message passing, although a key feature of components, is not subject to such interest. Instead, the semantics of message passing are fixed, or can, at best, be picked from a small set of fixed semantics. Examples are the remote procedure call mechanism, asynchronous message passing, broadcast and multicast, and atomic message send semantics.

These fixed semantics cannot be tailored to model application-specific interaction mechanisms. To model alternative interaction mechanisms, code for implementing these must be added to all implementations of components participating in the interaction, provided it is possible. It is even harder to abstract interaction mechanisms and reuse this abstraction. Concluding, a mechanism, which can implement tailored semantics for message passing is required to solve these problems. Such a solution should allow the application of component-oriented techniques to obtain extensibility and reusability. This is termed as the *fixed message passing semantics* problem.

2.2.5 Inheritance and Aggregations

Generally, behavioral composition of components is realized using inheritance and/or aggregation mechanisms.

Inheritance provides a compile-time *XOR-signature* composition of operations of components in a transitive way. An *XOR-signature* composition means that any inherited or self-defined operation of a component may be invoked exclusively and independently. A *transitive-composition* means that the inherited operations are automatically available

to the subclasses of the component. The pseudo variable *self* [3] is essential, and used to refer to the component which received the operation call.

In case of an aggregation, the composed component should aggregate the components that are needed to be composed. This is for example exemplified by various design patterns such as Bridge, State and Strategy [10]. A programming effort is necessary to implement *XOR-signature* composition such as declaring the necessary operations at the interface of the aggregating component and forwarding the calls on these operations to the aggregated components.

Inheritance and aggregation show different run-time and visibility characteristics. Both mechanisms have advantages and disadvantages. The main advantage of inheritance is that it provides a transitive reuse mechanism. Further inheritance is generally more efficient to implement than component aggregation. Disadvantages of inheritance are that inheritance cannot cope with run-time extensions and requires knowledge about the implementation of components. Further, in case of multiple inheritance name conflicts of the inherited operations may occur.

One important advantage of aggregation is that the aggregating component depends only on the interface operations of the aggregated components. This simplifies the design and improves the ease of reuse. Further, the aggregated components can be easily changed at run-time. Because of these advantages, most component models prefer aggregations to inheritance.

The main disadvantage aggregation is the lack of support for a transitive reuse mechanism. For example, sometimes in a distributed system the interface of a component cannot be fixed. Implementation improvements or migrating to a different platform may require extensions to the interface of a component. The Bridge and Strategy patterns [15] can be used to define components with dynamically changing implementations[4]. These patterns represent the alternative implementations as aggregated components, which can be changed at run-time. There are, however, a number of problems with this approach [3]. First, the aggregating component must declare all the operations explicitly and forward the calls to the aggregated components. This is an error-prone task. Inheritance allows all the inherited operations available transitively without necessarily declaring them. However, inheritance provides only a compile-time extension. Second, declaring operations at the interface of the aggregating component results in fixing the

[3] The pseudo variable *self* is called *this* in C++ and Java, and *current* in Eiffel.

[4] The Command pattern [10] can be considered more suitable in implementing components with changing interfaces. This pattern provides a limited reflection on messages. Reflection is discussed in subsection 2.2.7.

number of operations that a component can provide. Therefore, the operations of a component cannot be easily extended at run-time[5]. Finally, the aggregated components cannot access the aggregating component directly by invoking on *self*. Although the pseudo variable *self* can be simulated in the implementation of a component, the programmers may have to deal with two inconsistent pseudo variables: one defined for the components and the other defined by the implementation language, such as Java.

2.2.6 Delegation

Delegation is introduced as an alternative to inheritance [11]. In delegation, if a component cannot respond to a particular request of its client, then it delegates this request to one or more *designated* components. One of the designated components may execute the request on behalf of the delegating component. Further, the designated component can refer to the delegating component by calling on the pseudo variable *self*. Delegation is similar to inheritance; the designated component behaves like the super-class of the delegating component. Delegation can extend the interface of a component if the component delegates the requests to a another component, which extends the interface[6]. Delegation eliminates the need of declaring the interface operations explicitly. Further, compared to aggregation, delegation provides the pseudo-variable *self* so that the designated components can refer to their delegating component by invoking on *self*.

An additional advantage of delegation is to share a common behavior at the component level; the shared component may a have state, which can affect the shared behavior. Assume for example that in a distributed system multiple components provide the operation *checkSecurity*. All these components share the same implementation of *checkSecurity*. The implementation of *checkSecurity* refers to an access-list to verify invocations. It is desired that this access-list be encapsulated by the operation *checkSecurity*. Delegation can easily implement such a requirement. The shared designated object should then implement the operation *checkSecurity* and encapsulate the access-list. It is not easy to implement such a mechanism using inheritance because the operation *checkSecurity* must then be inherited from the super-class and the super-

[5] Inheritance suffers from the same problem. In strongly typed languages, generally a compile-time error is generated if, say component C_1 is replaced by an instance of its sub-class, say component C_2, and if an extended operation defined in C_2 is invoked on C_1. To eliminate these errors, generally typecasting is used.

[6] This is only possible if the adopted language does not generate typing errors, when the new operation is called.

class must also encapsulate the access-list. Classes are not optimized for storing and encapsulating state. This is termed as the *sharing behavior with state* problem [2]. Unfortunately only a few languages support delegation [18].

One disadvantage of current implementations of delegation is that they cannot enable or disable the delegation process, for example, based on a condition of the delegating component. This may be necessary if the implementation of a component has to be adapted based on certain conditions. For example, in distributed systems a *conditional delegation* mechanism could be useful in adapting the protocols based on the quality of service requirements. Although the Bridge and Strategy patterns provide a similar functionality, they are aggregation-based and therefore do not provide the advantages of delegation as discussed in this subsection. In a conditional delegation mechanism, the designated components could implement the alternative protocols, and the condition of the delegating component would enable the designated component which implements the most suitable procol. This is called the *lack of support for dynamic composition* problem.

2.2.7 System Interface and Layering

The interface of middleware systems must be fixed to achieve compatibility and portability. As stated in the previous sections, however, the application domain of the middleware technology grows steadily. Generally, this requires extensions to the middleware services. We term this as the *unmatched system functions* problem.

The application programmer may deal with this problem in three ways:

- Implementing the service at the application level. This approach causes a repetitive re-implementation of the common service for every similar application.
- Implementing the service as an *application service*. Existing middleware systems provide certain application services that can be explicitly called by applications if needed. These application services are defined at the same level as applications. This approach, is therefore, not desirable if the service must be transparent to the application program.
- Implementing the service within the middleware. This approach is difficult to realize because generally middleware systems are *closed systems* and do not provide facilities to extend them.

One of the reasons why middleware systems are closed systems is due to transparency of certain services. Middleware systems are organized in layers

and layers provide transparency in that certain aspects of a layer are invisible to the "higher-level" layers. Typical examples are location transparency, replication transparency, failure transparency, etc.

Although transparency of certain services eases application programming, it works against extensibility for two reasons: First, to be able to implement new transparent services, it must be possible to introduce new transparent layers into the system without re-defining the existing ones. For example, certain systems may require a dedicated transparent security layer, which might not be foreseen before. In case of a closed middleware system, however, this is not possible. Second, to be able to cope with the evolving application demands, it must be possible to adapt the services of a layer in a controlled manner. Transparency of certain aspects, however, may hinder the adaptation of certain services.

In the literature reflective systems are proposed to "weaken transparency" in a controlled manner [19]. A reflective system is a system which incorporates models representing (aspects of) itself. This self-representation is *casually connected* to the reflected entity, and therefore, makes it possible for the system to answer questions about itself and supports actions on itself. Reflective computation is the behavior exhibited by a reflective system.

The term reflection was introduced by [16] as a technique to structure and organize self-modifying procedures and functions. In [12] reflection was applied within the context of object-orientation. A considerable amount of work has been done in reflection techniques, for example, in concurrent computation, distributed system structuring and middleware, programming language design, real-time systems, and in network design [20][6]. Conventional component models do not provide adequate support for reflective system development. This is termed as the *lack of support for reflection* problem.

Recently, several attempts have been made to provide message reflection within a middleware. The so-called *portable interceptors* can be inserted into a middleware, which enables the programmers to access, test and modify the messages. Reflective computation is an active research area and there are a number of open questions:

- Which aspects of components must be reflected? For example the portable interceptors in CORBA are only defined for a limited set of interactions within the middleware. It is currently not clear how many interceptors are necessary at which levels. Moreover, intercepting provides only a limited degree of reflection.
- What are the suitable abstraction techniques? How can reflection be managed? For example the main abstraction of portable interceptors is

the representation of messages. This provides only a limited reflective capability.

- What are the suitable techniques to specify and enforce causal connections among the reflective levels? For example in the portable interceptor approach, this is basically defined at the level of manipulating messages.
- What is the affect of evolution to the reflective part of components? Can reflective parts easily evolve as well?
- What kinds of compositional mechanisms must be provided at the reflective layer? This is an important issue if multiple dependent properties are reflected. Since these properties are related to each other, suitable compositional mechanisms must be provided at the reflective level as well.
- If a certain property is shared among multiple components and/or levels, how can this property be reflected?

Especially the last three items pose problems and can not be solved adequately using the current reflection techniques. For example, in the portable-interceptors approach, it is not clear how the processing of reflected messages at the reflective-level can be composed. Consider for example the quality of service properties of a typical middleware system. These properties generally refer to multiple components in multiple layers and therefore cannot be easily captured by reflecting a single message, component or a layer.

3. THE COMPUTER SCIENCE SUB-DOMAINS

Based on the assumptions given in the previous section, we have selected the following sub-domains: application generators, concurrency and synchronization, constraint Systems, control systems, distributed processing and real-time systems. Obviously, practical software systems may include several of these sub-domains.

3.1 Application Generators

Figure 3 depicts a typical application generator (AG) architecture. AGs are used to construct software in a particular domain. AGs usually adopt a high level specification language in which the application can be described. The application domain needs to be well understood to allow applications to be described at a higher level of abstraction. AGs incorporate some default information in their application domains and thereby let the programmers

concentrate on the specific aspects of their programs. Based on the user's specifications and default knowledge, AGs construct the application in an executable (general purpose) language.

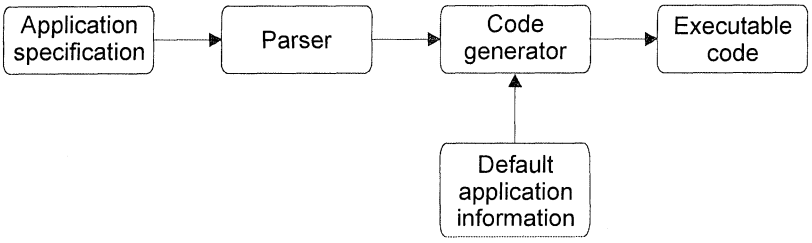


Figure 3: A block diagram of application generators

In middleware systems, AGs are used to generate stubs; stubs are software modules that link application-level functions to the functions of the middleware layer in a transparent manner.

In principle, AGs can be used in distributed systems as a general mechanism to hide certain implementation aspects of applications. For example, the application programmer may express his/her specific security needs in a *directive file*. A suitable AG can interpret this specification and bind the application to some dedicated services in a transparent way. The subsections 3.1.1 and 3.1.2 describe the important aspects of AGs.

3.1.1 Problem Domain Specification

The specification language of an AG must be expressive enough to describe all the important features in its domain.

Problem domain specification and implementation by itself is a complete analysis and design problem. Therefore, depending on the domain, the designer of an AG may suffer one or more of the problems listed in this chapter.

Secondly, while designing and reusing AGs, designers may experience the *arbitrary composition* problem[7], which is explained in the following:

Assume for example that we would like to generate the interaction protocol of a component from a specification language. If the protocol has various different versions and/or the future extensions are likely to occur, we may want to have composition mechanisms for this specification language. Reusing and composing these specifications using the standard class inheritance or component composition mechanisms may not be appropriate

[7] This is similar to the *arbitrary inheritance* problem which was described in [2][3].

since specifications may include specific constructs which cannot be supported by the standard inheritance and composition mechanisms.

3.1.2 Code Generation

The code generator requires that all the necessary details be provided in the input specifications. The generated code must be complete enough to execute on current middleware platforms in an efficient way. This requires realizable_and precise_models.

On the other hand, it is not desired to overload the programmer with too many details. Assume, for example that we like to optimize the remote accesses by considering the application characteristics, such as access rates, size of data, as much as possible. If the user does not give any details, it may be difficult for the AG to generate an efficient code. In general, the performance problem of the generated application cannot be improved only by the compiler of the generated code. There is a need for problem-domain specific optimization techniques, which have to be provided by the code generator of the AG.

3.2 Concurrency and Synchronization

Concurrent processing can be defined as a parallel or time-multiplexed execution of one or more operations, which can be, but are not necessarily, data-dependent. Whenever the execution of one operation is started before the execution of another operation is completed, the two executions are concurrent. The basic aspects in designing concurrent systems are presented in subsections 3.2.1 and 3.2.2.

3.2.1 Creating Concurrency

This can be done either explicitly through special constructs (e.g. *fork* [8]), or implicitly, as the result of certain message passing semantics. For example, when applying asynchronous message passing, the thread that initiates the call proceeds concurrently with the thread executing the call. The creation of new active objects is an additional possibility for creating concurrency.

Although some methods and programming languages provide a variety of message sending constructs, in general the semantics of message sending are fixed and cannot be tailored to particular needs. In section 2.2.4, this was defined as *fixed message passing semantics* problem.

3.2.2 Synchronization of Concurrent Executions

The activities of concurrently executing processes can be totally independent, but they may also interact at certain times. This interaction may involve the exchange of data, such as producer-consumer interaction, or can be *pure synchronization* between two or more processes, often with the aim of safely sharing resources.

Three approaches for integrating concurrency and component models can be distinguished:

- *Orthogonal approach*: Components and threads are completely independent, creation of threads and synchronization are achieved through special statements, such as the conventional *forks* and *semaphores*.
- *Homogenous approach*: Every component is considered to be active, and takes care of its own synchronization; components are the unit of concurrency.
- *Heterogeneous approach*: Adopts both passive and active components. Passive components do not perform any synchronization on incoming requests.

The homogeneous approach is well integrated with the component model. However, the combination of component composition with synchronization introduces new problems[8]. This is termed as the *composition versus synchronization* problem.

Assume for example that the message invocation performance of a middleware system has to be improved by dynamically switching between various protocols. For instance, if a client component invokes on its server only once, then relatively, the TCP/IP protocol is considered as the most efficient implementation. However, if the same client component starts to invoke on multiple servers repeatedly, then the multicast protocol is considered the more efficient. Studies show that [15] in current middleware systems, however, switching from one protocol to the other at run-time is generally not possible. This is because, the synchronization action is distributed to various components and layers. As a result, new protocols cannot be easily incorporated at run-time without replacing a considerable part of the middleware system.

[8] This is similar to the *inheritance anomaly* problem [13]. An extensive discussion of these anomalies is given in [4].

3.3 Constraint Systems

In constraint systems applications are modeled using components and relations between these components that need to be enforced. These relations, in general, are constraints on the attribute values of the components and expressed in formulas. As constraints are often related to each other, changing the value of one attribute can have a considerable effect on the attributes of other components.

Assume for example that a distributed geographic information system has to be designed. This system allows creation, modification and deletion of geographic data by multiple users. Geographic data is constraint by the topology. In addition, the physical characteristics the artifacts, such as houses and roads have to be considered. The system has to maintain the constraints when certain artifacts are created or modified.

Subsections 3.3.1 to 3.3.3 present the important aspects of designing constraint systems.

3.3.1 Constraint Enforcement Strategies

The constraint enforcement system must determine which component's attribute needs to be changed when a constraint is violated. Assume that the constraint is expressed as $x + y = z$. Here, if z is the independent variable, then the constraint can be enforced by updating the dependant variables x and y. If, however, none of the variables are independent, then the problem becomes even more complicated, since the constraint system must adopt a more complicated constraint enforcement strategy.

Since constraint enforcement among components can be defined as a coordination activity, components must represent the coordinated behavior explicitly. The Observer and Mediator patterns [10] for example, can be used to detect a change and enforce the constraints, respectively. The Mediator pattern encapsulates the implementation of the constraint enforcement strategy.

The designers can implement the constraint enforcement strategy at the application, application service or middleware levels. Here, the issues to be concerned are similar to the ones that were discussed in 2.2.7.

An important question here is how to implement the constraint enforcement strategy in a reusable and extensible manner. Generally, constraint enforcement is implemented by sending the necessary messages to the participating components. For example, the interaction pattern can be implemented by using the Mediator pattern. The interaction is mainly based on *message send* semantics, where the *mediator component* transmits messages one by one to the so-called *colleague components*. We consider

the message send model as being too low-level, because it can only specify communications that involve two partner components at a time and its semantics cannot be easily extended. Mechanisms like inheritance, aggregation and delegation only support the construction and behavior of components but not the abstraction of communication among components. These mechanisms therefore fail in abstracting patterns of messages and larger scale synchronization involving more than just a pair of components. For example, it is not trivial to extend the mediator component for the purpose of extending the constraint enforcement strategy. This is termed as the *lack of support for coordinated behavior* problem.

3.3.2 Conflicting Constraints

Just as each constraint can be related to multiple components, each component can be related to multiple constraints. This can result in conflicting constraints when the value ranges resulting from the constraints are not overlapping. One solution is to assign priorities to constraints and let the constraint with the highest priority be enforced.

If the constraint system is implemented as an application service or provided transparently by the middleware system, conflict resolution strategies may not be replaced or extended easily. This was termed in section 2.2.7 as the *unmatched system functions* problem.

3.3.3 Reuse of Constraints

The current component models do not support the use of constraints as an explicit feature. If constraints are added to components as attributes, then it may be difficult to reuse and extend them by using the current composition mechanisms. In section 3.1.1, this was termed the *arbitrary composition* problem.

Note that if the constraint system has to be generated automatically, then it is a special kind of application generator and includes the aspects of application generator design.

3.4 Control Systems

As shown by Figure 4, a (feedback) *control system* includes sensors and actuators to influence (or control) the behavior of the *controlled system.* Each measured value of the controlled system is compared with its associated desired value and based on the discrepancies the parameters for the related actuators are calculated and engaged. The controlled system may be influenced by several other sources besides the control system. In

addition to the controlled system, modeling these sources and reacting to them properly can dramatically improve the quality of control. In short, a control system usually adopts models for the controlled environment, actuators, sensors and the controlling algorithms. Note that if a software system controls another software system, all the elements of a control system are implemented in software.

Subsections 3.4.1 to 3.4.3 present the important aspects of designing control systems.

3.4.1 Control Specifications and Algorithms

A large diversity of systems can be controlled, and therefore, controlled systems can have arbitrary complexity. Depending on the models associated with the controlled system, the analysis and design of a control system may include one or more of the problems that are identified in this chapter.

Assume for example that the quality of service parameters of a distributed system has to be controlled using the architecture given in Figure 4. Here, the model of the controlled system must express the actual quality of service parameters of the distributed system under consideration. The reference model must be expressive enough to represent the desired parameters. The controlling algorithms must have the knowledge of adapting the distributed system in such a way that the properties are adjusted in the right manner. The actuator must have the capability of executing the adjustments.

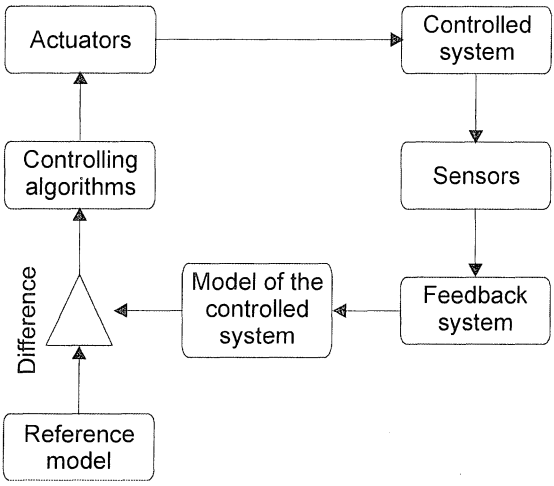


Figure 4: Block diagram of a control system

3.4.2 Sensing, Monitoring and Feedback

The control system must sense, monitor and control the controlled system. There is a meta-level relation between the controlled system and the control system. To effectively represent these meta relations, the underlying object-oriented model must provide a *reflection mechanism*.

Assume for example that the quality of service parameters of a middleware system has to be controlled. In this case, the controlled system is the middleware and the control system is responsible for the quality of service management. The quality of service parameters of a middleware can be adjusted, for example, by reconfiguring the middleware components. To be able to control the middleware, however, certain characteristics of the middleware must be measurable by the control system. In case of a closed middleware system, measuring the necessary parameters maybe too difficult or even impossible. In section 2.2.7, this was termed as the *lack of support for reflection* problem.

3.4.3 Coordinated Behavior

Complex and large control systems often consist of several distinct units such as controlling algorithms, actuators, sensors, and/or low-level sub-control systems. Depending on the architecture and/or controlling algorithms, the units of a control system coordinate together to keep the controlled system consistent with respect to its pre-defined specification.

As an example, consider again the implementation of the quality of service management system. Here, the measurements must be collected from the relevant and possibly distributed components. Similarly, adjustments must be realized on these components. This requires implementation of well-defined interaction mechanisms. If the interactions must be extended, the *lack of support for coordinated behavior* problem explained in section 3.3.1 can be experienced.

3.5 Distribution

A distributed computer system consists of multiple, cooperating, autonomous computer systems (nodes) that are interconnected by a communication network, and a middleware system to integrate such an interconnected system into a logical entity. In subsections 3.5.1 to 3.5.8, we present eight important aspects of distributed system design:

3.5.1 Remote Invocations

In a distributed system, components are scattered over the nodes of the system, and components on different nodes may need to communicate using the so-called remote invocations. The required semantics of remote invocations may differ according to the circumstances, although the remote procedure call model is used most frequently. In the remote procedure call model, the client component waits until the server component explicitly *returns* the result of the call. The remote procedure call semantics can be too restrictive for certain applications. In section 2.2.4, this was termed as the *fixed message passing semantics* problem.

3.5.2 Transparency

It is generally claimed that most of the activities within a distributed system must be transparent to the users/programmers of the system. The various possible sorts of transparency that can be added to distributed systems are summarized below:

Access Transparency: Whether or not there is any perceived difference of procedure in using a local resource or a remote resource.

Replication Transparency: Whether or not replication of components is visible.

Location Transparency: Whether or not access to (or control of) a resource is dependent on that resource of being at a particular location, held exactly one location, or distributed over several locations.

Failure Transparency: Whether or not failures that occur in system components affect the overall functioning of the system.

Although transparency makes it easy to write programs, there are some negative consequences, such as performance. In case of remote sharing, for example, performance can be negatively affected if topology information is not taken into account. Similarly, efficient and proper exception handling and increasing availability require more or less topology information. Moreover, in section 2.2.7, we discussed the negative affects of transparency to extending the services provided by the middleware system. Within this context two problems were discussed: First, the *unmatched system functions* problem, which can be experienced if the services of the middleware do not match the needs of the applications. Second, the *lack of support for reflection* problem, which can be experienced if the transparent middleware services are required to be modified or new services have to be introduced.

3.5.3 Resource Sharing

For sharing resources, generally a client-server model is used; a (server) component encapsulates the shared resource and serves requests from other (client) components. This model is sufficiently supported by the current component models.

3.5.4 Distributed Algorithms

Distributed algorithms are frequently required in distributed systems, not only for the implementation of middleware system level functions, but also for the realization of application level software systems. A distributed algorithm involves a number of components that exchange messages with each other.

In current component models, interaction code is likely to spread over all the participating components. In section 3.3.1, we discussed the difficulties in reusing and extending interaction code among components. This was referred to as the *lack of support for coordinated behavior* problem.

3.5.5 Distributed Concurrency Control

In a distributed system, data is partitioned and/or replicated over the nodes of the system, and commonly shared by multiple nodes. One of the major problems is to keep the data consistent. The most common mechanism to address this is the *transaction mechanism*. To guard data from inconsistencies in the presence of hardware or system failures, or when a certain sequence of actions have to be executed in an indivisible way, then the transaction mechanism can be adopted. A transaction is a sequence of events that are *serializable* and *indivisible*.

In the literature, many different transaction implementation techniques are presented. Implementations generally differ from each other with respect to their serialization and/or recovery characteristics. The suitability of an implementation technique may depend on the application, system and/or the data structure. It is therefore difficult to select a single implementation as a middleware service for all applications. In section 2.2.7, this was termed as the *unmatched system functions* problem.

If the application programmer requires choosing the implementation of middleware transaction services, and if the transaction services are transparent, the *lack of support for reflection* problem can be experienced. If the middleware has to automatically select the best transaction implementation in a transparent way, then this is a control system design problem as discussed in section 3.4. In this case, defining the right model for

the controlled middleware system and the right criteria for selecting the optimal transaction implementation can be a difficult task.

3.5.6 Recovery

Recovery is a facility for ensuring that no information is lost even in the occurrence of software and hardware failures. To cope with hardware failures, there must be a so-called *stable storage*, which is guaranteed to remain unaffected by failures. Recovery is usually part of the transaction mechanism; when a transaction succeeds, the new state is guaranteed to be saved, when the transaction is aborted half-way, the system is recovered to the state before the start of the transaction. The problems that designers may experience are similar to the ones of transactions.

3.5.7 Security

The security of an information system becomes more critical especially when the processing nodes are distributed. *Access-control lists* and *capabilities* are two well-known techniques [9]. Cryptographic techniques can be integrated with the system to reduce the vulnerability [14]. Security mechanisms must be superimposed upon other distributed system functions. Generally, this means introducing transparent services to the middleware. In section 2.2.7, this was defined as the *lack of support for reflection* problem.

3.5.8 Layered Communication Protocols

Distributed systems, in general, are structured in terms of layers. Functionally, each layer communicates with its peer-level layer, although physical data exchange occurs with the adjacent layers. In the section 2.2.7, the problems associated with designing layers were identified as the *unmatched system functions* and the *lack of support for reflection* problems.

3.6 Real-time

A real-time (RT) system is a system that is required to respond to external events within a predetermined time interval. RT systems are often classified into *hard-* and *soft* real-time systems. In hard-RT systems, violation of the time constraints will result in serious, possibly catastrophic damage to the system and its environment. Soft-RT systems aim at fulfilling the time constraints as much as possible. One class of soft-RT systems are *statistical* RT systems where average response times are required. For hard-

RT systems several analysis techniques have been developed to guarantee the timely behavior of the system under all circumstances.

In subsections 3.6.1 to 3.6.3, the important aspects of RT systems design are presented.

3.6.1 Real-time Specifications

In current component models, RT specifications may conflict with the composition mechanism. In case components with real-time behavior are required to be extended or modified through component composition, the designers may be obliged to redefine some or all of the features of the predefined components although this may be intuitively unnecessary. This is referred to as the *composition versus RT specifications* problem[9].

RT specifications can also be defined for the coordinated actions of components, requiring an explicit representation of component coordination. The problem that is related to coordination was discussed in subsection 3.3.1 and was termed as the *lack of support for coordinated behavior* problem.

3.6.2 Dynamicity in RT Specifications

The current RT specification techniques lack flexibility in the association of RT specifications to components. In general, only the server component associates RT specifications with its operations, which are fixed during the life time of this component. Especially for soft-RT systems, it may be desirable to select an optimal RT specification for an operation from various alternatives. Moreover, client components are also not able to define time-intervals on their calls to server components. These are similar to the problems that were discussed in subsections 2.2.3 and 2.2.6 and were respectively termed as the *multiple-views* and *lack of support for dynamic composition* problems.

3.6.3 Temporal Behavior Analysis

Several algorithms for determining the temporal behavior of a RT system have been defined. As the problem itself is, at least NP-complete, the useful algorithms are heuristic algorithms determining an upper bound. There problems were discussed in the literature and termed as *schedulability analysis*. Efficient algorithms have to be defined for the analysis of RT component models. Other important aspect for soft-RT systems is exception handling in case the timing constraints can not be fulfilled.

[9] This is similar to the *real-time specifications inheritance anomaly* problem [4]

4. CONCLUSIONS

Constructing software systems from components has several advantages such as managing complexity and run-time adaptability. There are, however, a number of obstacles that software engineers may experience while designing systems from components. In general, it is possible to classify these obstacles as *lack of expression*, *decomposition* and/or *composition* problems. In this chapter, these general problems are further specialized into 11 problems and are classified according to the aspects of domains of applications. By using the guidelines presented in this chapter, the software engineers may identify the potential problems first by identifying the domains of their applications, and then by considering the aspects of each domain.

Various techniques have been introduced in this book to overcome some of the obstacles presented in this chapter.

The *composition* and *decomposition* problems in design are addressed by the architecture design methods presented in chapters 3, 4 and 5.

The *lack of expression* is a general problem and addressed by parts 2 and 3 of this book. Every chapter addresses this problem within a specific problem context. For example chapter 6 presents a set of abstractions which are capable of expressing message-oriented architectures, chapter 7 presents a logic meta-programming language to capture architectural constraints, chapter 9 presents basic language mechanisms to express a large category of component composition mechanisms, etc. Chapters 10, 11 and 12 aim at enhancing the expression power of current languages by providing effective mechanisms for separation and composition of concerns.

The *arbitrary composition* problem is not directly addressed in this book. In our related research activity, we addressed this problem partially within the context of designing parser generators. We introduced the so-called *grammar inheritance mechanism*, in which sub-grammars may be inherited from the super-grammars [1]. This technique is especially suitable to reusing the grammars of specifications.

ACKNOWLEDGEMENTS

This research has been supported and funded by various organizations including Siemens-Nixdorf Software Center, the Dutch Ministry of Economical affairs under the SENTER program, the Dutch Organization for

Scientific Research (NWO, 'Inconsistency management in the requirements analysis phase' project), the AMIDST project, and by the IST Project 1999-14191 EASYCOMP.

5. REFERENCES

1. M. Akşit, R. Mostert and B. Haverkort. Compiler Generation Based on Grammar Inheritance. Memoranda Informatica 90-07, University of Twente, February 1990.
2. M. Akşit and L. Bergmans. Obstacles in Object-Oriented Software Development. *In Proceedings OOPSLA '92*, ACM SIGPPLAN Notices, Vol. 27, No. 10, pp. 341-358, October 1992.
3. M. Akşit, B. Tekinerdoğan, F. Marcelloni and L. Bergmans. Deriving Object-Oriented Frameworks from Domain Knowledge. In *Building Application Frameworks: Object-Oriented Foundations of Framework Design*, M. Fayad, D. Schmidt, R. Johnson (Eds.), John Wiley & Sons Inc., pp. 169-198, 1999.
4. L. Bergmans and M. Akşit. Composing Synchronisation and Real-Time Constraints. In *Journal of Parallel and Distributed Computing*, Vol. 36, No. 1, pp. 32-52, 1996.
5. A. Burggraaf, Solving Modelling Problems of CORBA Using Composition Filters. MSc. thesis, Dept. of Computer Science, University of Twente, August 1997.
6. P. Cointe (Ed.). *Meta-Architectures and Reflection*. Springer Verlag LNCS 1616, St Malo, May 1999.
7. S. de Bruijn. Composable Objects with Multiple Views and Layering. MSc. thesis, Dept. of Computer Science, University of Twente, March 1998.
8. J. B. Dennis and E. C. van Hoorn. Programming Semantics for Multiprogrammed Computations. In *Communications of the ACM*, Vol. 9, No. 3, pp. 143-155, March 1966.
9. R. S. Fabry. Capability-Based Addressing. In *Communication of the ACM*, Vol. 17, No. 7, pp. 403-412, July 1974.
10. E. Gamma, R. Helm, R. Johnson and J. Vlissides. *Design Patterns: Elements of Reusable Software*. Addison Wesley, 1995.
11. H. Lieberman. Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems. In *Proceedings of OOPSLA'86*, ACM SIGPLAN Notices, Vol. 21, No. 11, pp. 214-223, November 1996.
12. P. Maes. Concepts and Experiments in Computational Reflection. In *Proceedings OOPSLA'87*, ACM SIGPLAN Notices, Vol. 22, No. 12, pp. 147-155, December 1987.
13. S. Matsuoka and A. Yonezawa. Inheritance Anomaly in Object-Oriented Concurrent Programming Languages, In *Research Directions in Concurrent Object-Oriented Programming*, G. Agha, P. Wegner and A. Yonezawa (Eds.), MIT Press, Cambridge, MA, pp. 107-150, October 1993.
14. S. Mullender and A. Tanenbaum. Protection and Resource Control in Distributed Operating Systems. In *Computer Networks*, No. 8, pp. 421-432,1984.
15. M. Sinderen (Ed.) *Application of Middleware for Services in Telematics (AMIDST) Project*, CTIT, http://amidst.ctit.utwente.nl/, 1999.
16 B. C. Smith. Reflection and Semantics in a Procedural Language. MIT-LCS-TR-272, Mass. Inst. of Tech. Cambridge, MA, January 1982.
17. C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

18. D. Ungar and R. B. Smith. Self: The Power of Simplicity, In *Proceedings OOPSLA'87*, ACM SIGPLAN Notices, Vol. 22, No. 12, pp. 227-242, December 1987.
19. Y. Yokote. The Apertos Reflective Operating System: The Concept and Its Implementation. In *Proceedings of the OOPSLA'92*, ACM SIGPLAN Notices, Vol. 27. No.10, pp. 414-434,October 1992
20 A. Yonezawa (Ed.). Reflection and Meta Level Architecture. Proceedings of IMSA'92, Tokyo, November 1992.

APPENDIX DEFINITION OF THE OBSTACLES

- *Arbitrary composition*: was defined in subsection 3.1.1 as the difficulty in composing components if some or all aspects of components are generated from specifications and if the specifications cannot be composed by using the composition mechanism of the component model. This problem can be experienced mainly in designing application generators and constraint systems.
- *Composition*: was introduced in subsection 1.2 as the difficulty in creating new components by reusing simpler components. Effective composition demands two complementary characteristics from the component model. First, it requires as much as possible to reuse the existing components without modifying them. Second, the adopted composition mechanism must be suitable in utilizing the existing components in creating new components.
- *Composition versus RT specifications*: was defined in subsection 3.6.1 as the difficulty in reusing or extending RT specifications of components. This problem can be experienced in designing components with RT behavior.
- *Composition versus synchronization*: was defined in subsection 3.2.2 as the difficulty in reusing or extending synchronization code of components. This problem can be experienced in designing concurrent components with explicit synchronization.
- *Decomposition problem*: was introduced in subsection 1.2 as the difficulty in defining autonomous components in solving a given problem due to the complexity of the problem.
- *Excessive type declarations*: was defined in subsection 2.2.2 as a result of the necessity to declare a separate type module for every consistent composition of components.
- *Fixed message passing semantics*: was defined in subsection 2.2.4 as the difficulty of defining and reusing tailored message passing semantics of interacting components.

- *Lack of expression power*: was introduced in subsection 1.2 as the difficulty in expressing software directly by using the features of the adopted language.
- *Lack of support for coordinated behavior*: was introduced in subsection 3.3.1 as the difficulty of representing and reusing coordination among components especially if the coordination is implemented as multiple messages.
- *Lack of support for dynamic composition*: was introduced in subsection 2.2.6 as the difficulty of adapting composition structures (such as inheritance or delegation) at run-time.
- *Lack of support for reflection*: was introduced in subsection 2.2.7 as the difficulty of redefining the primitive or transparent features of the system.
- *Multiple views*: was introduced in subsection 2.2.3 as the difficulty of adapting the interface of a component based on its context.
- *Sharing behavior with state*: was introduced in subsection 2.2.6 as the difficulty of sharing a common behavior among components if the behavior is affected by a common state.
- *Unmatched system functions*: was introduced in subsection 2.2.7 as the mismatch of application needs and the functions provided by the system layer.

PART 2

ARCHITECTURES

Chapter 3

COMPONENT-BASED ARCHITECTING FOR DISTRIBUTED REAL-TIME SYSTEMS

How to achieve composability?

Dieter K. Hammer
Department of Computing Science, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, the Netherlands. email : hammer@win.tue.nl

Keywords: Architecture, architectural views, components, composability, dependability, design space, design method, distributed real-time systems, embedded systems, emergent properties, interfaces, non-functional constraints, stakeholders, system behavior

Abstract: In systems- and software architecting, architecture can be viewed as a high-level design that supports the construction of ICT-systems. Starting from a list of general requirements, the first part of this chapter gives an overview of the dimensions of such a design. In addition, the various, often contradicting, architectural views that are relevant for the various stakeholders are discussed. Special emphasis is given to the modeling of the system behavior and the dependability constraints. The second part of this chapter summarizes the requirements that binary components must fulfill in order to be composable in the context of dependable distributed real-time systems. Thereby, the emphasis is on timeliness and reliability. It is argued that in order to achieve composability, resource requirements and non-functional properties are of equal importance as functionality. In addition, the architectural styles that govern the interaction of components with their environment must be specified. A method for constructing the collective behavior of a set of components and achieving composability is sketched and demonstrated by means of an example.

1. THE MANY ASPECTS OF AN ARCHITECTURE

Architectures are increasingly used to guide the development and production of complex industrial ICT-products and to improve the related decision processes. Architectures play an important role in balancing the demands of the market and the customers, the technical solutions and the characteristics and capabilities of an organization as e.g. described in [8]. It is also recognized that different stakeholders (like development-, production and service engineers, development-, product- and production managers, sales people and users/customers) have different needs and consequently use different views on an architecture.

In the literature as well as in practice, there is, however, no commonly agreed view on these issues in terms of the definition of an architecture, the description and evaluation of an architecture and the architecting process. The main problems regarding the concept of an architecture can be summarized as follows:

a) Current architectures are often developed in an ad-hoc way. As a consequence, the system development process is time-consuming and not flexible.
b) Architectures are often developed for single systems and do not well support the variety associated with system families and product platforms.
c) Architectures are often defined without giving enough attention to the product development process. In particular in the upper lifecycle phases of ICT-products, i.e. in the communication process with customers and users, the concept of an architecture is vague and ambiguous.
d) Current architectures concentrate too much on one particular use (system-, hardware- and software design, system configuration, etc.) and do not support multiple consistent views that support the activities of the various stakeholders mentioned above.
e) Present architectures concentrate too much on the structure of the system and do not adequately deal with the dynamic aspects, i.e. the behavior. The emphasis is on the modularization of the system and the definition of the interfaces. Even if the various interaction channels are modeled, there are no restrictions of the actual interaction patterns that can occur at runtime.
f) There is not enough attention for the non-functional aspects of an architecture like dependability (performance, timeliness, reliability, availability, safety, security and robustness as defined in [21]) and other X-abilities (modularity, composability, scalability, maintainability, adaptability, extensibility, openness, interoperability, portability,

reusability, etc.). Since we also miss adequate development methodologies that support the design of these system dimensions during all development phases, these considerations enter the construction of an ICT-system often only during the implementation phase.

g) Current architectures do not support requirements tracing, i.e. they do not support a mechanism to document which architectural component, relation or constraint implements which requirement and vice versa. The same problem occurs for the tracing between the features of an architecture and their implementation.

Section 1 of this chapter tries to identify the issues that are important for the development of an architecture and to define relevant research questions. The first two subsections deal with the definition and the purpose of an architecture. Subsection 1.3 introduces a number of general starting points for the design of an architecture and elaborates on the modeling of the behavior of ICT-systems. The relevant dimensions of an architecture are discussed in subsection 1.4. Subsection 1.5 identifies the most important stakeholders of an architecture and summarizes the corresponding architectural views and their relation with the various design dimensions.

Section 2 discusses the requirements for achieving composability in Event-Triggered Architectures (ETA's). The emphasis is on dependable distributed real-time systems and especially on timeliness and reliability. In subsection 2.2, the extensions of the component interfaces necessary for achieving composability are described. Subsection 2.3 summarizes the development method underlying this discussion. Subsection 2.4 illustrates the proposed approach by means of an example.

Finally, a number of conclusions are drawn and important directions for future research are indicated in section 3.

1.1 What is Architecture?

The essence of architecting is the structuring of the construction and the behavior of an ICT-system. In this way the possible design alternatives are restricted and the various people involved in the design, production and maintenance of an ICT-system are supported in making sensible decisions. As in any design activity, also for defining architecture the key issues are balancing of the often contradictory requirements, compromising between extremes, and, of course, also elegance.

At the moment, there is no generally agreed definition of architecture. In the literature, various often very general descriptions can be found. Rechtin [26] e.g. defines an "overall system architecture" as the structure of a system, its functions and its environment together with the structure of the

process by which it will be built and operated. Dewayne and Wolf [25] concentrate on software architectures and define them as a combination of elements, form and rationale. *Elements* can be processing-, data- or connecting elements. *Form* includes the importance of properties and relationships, the constraints on architectural elements and the constraints on the relations between architectural elements. Finally, the *rationale* gives motivations for the choice of a particular architectural style, the choice of the elements and the form. Bass et al. [1] define an architecture as "the structure or structures of the system, which comprise components, the externally visible properties of those components, and the relationship among them".

An architecture can be considered as a common high-level model or design that supports the various disciplines and stakeholders in

a) taking design decisions in a multi-dimensional design space,
b) defining their designs in a multi-disciplinary environment and
c) communicating their solutions.

Note that not all parts of architecture need to be defined at the same level of abstraction. Usually the architecture can be considered as high-level design but parts like interfaces and protocols can even be defined at implementation level. Of course, the different parts of architecture should fit into one consistent framework. The level of abstraction of the architectural model depends on the level of detail at which the various activities of the early project phases have to be performed, e.g. on the envisaged degree of concurrent engineering.

This chapter abstracts from the particular design or project to be supported by architecture. In addition, no distinction is made between systems- or software architecting.

Already Dewayne and Wolf [25] consider an architecture as one step in the construction process of a system. Since the classical distinction between an implementation-independent conceptual model (defining the what) and an implementation model (defining the how) is obsolete, this construction process is not considered in terms of a classical waterfall model (e.g. requirements, architecture, specification, design and implementation) but as a continuous process of top-down refinement. This view is more consistent with the practice of e.g. object-based methodologies where components can be identified at various levels of refinement and every component consists of a specification and a design that identifies the sub-components and their relations.

As already mentioned, most notions of an architecture concentrate too much on the structure of a design and do not adequately deal with its behavior. A typical example of this restricted view is given in an early paper

of Shaw [30] that concentrates on abstract data types. Dewayne and Wolf [25] go one step further and distinguish between a *process* view that concentrates on the data flow through the processing elements, a *data* view whose emphasis is on the processing flow and the *interdependence* of processing and data upon the interconnections between the processors. This approach is very similar to the dataflow paradigm that can e.g. be found in SA/SD (Structured Analysis/Structured Design) and some object-oriented methods.

Only recent definitions like the one given by Bass et al. [1] pay equal attention to structure and behavior. These authors define the architecture of a program or ICT-system as "the structure or structures of the system, which comprises software components, the externally visible properties of those components, and the relationships among them". Thereby, relationship refers to both the static interconnection between components by channels and the dynamic interaction. Unfortunately, this approach to architecting is still not widely used in industry.

1.2 What is Architecture Good for?

The overall aim of architecture is the improvement of the quality of the ICT-system and its product development process. Key factors for the success of a project are the emphasis on the early development phases, the interdependence between product and process, and the interdisciplinary nature of decision making by the architect(s) in cooperation with the various stakeholders. In particular, an architecture has the following important advantages:

a) It establishes a framework for satisfying the requirements.
b) It forms a technical basis for the subsequent refinement of the system.
c) It restricts the design alternatives by enforcing a particular architectural style (e.g. a central or a distributed architecture, an event-driven or a time-driven architecture, etc.) and taking important design decisions.
d) It supports requirements tracing, i.e. it provides a mechanism to document which architectural component, relation or constraint implements which requirement and vice versa.
e) In addition, it supports implementation tracing, i.e. the tracing between the features of an architecture and their implementation.
f) It forms a basis for reducing risks by supporting the various activities that have to be performed during the early development phases. Among the most important are
   – An evaluation in order to ensure the consistency and quality of the architecture,

  - a feasibility study in order to make sure that a system can be made within the given constraints and
  - an assessment of the threats and risks a project is exposed to.

g) It supports the answering of "what-if" questions that are e.g. related to different uses of the product and its future evolution.

h) It forms not only a basis for designing a system but also for designing the related engineering-, production-, service- and maintenance processes.

i) It forms the managerial basis for cost estimation, project management and configuration management. In particular, an architecture should support
  - project planning,
  - concurrent engineering in general,
  - hardware/software codesign in particular,
  - configuration management during the development phase and
  - configuration selection and management with respect to the market and the customers.

j) It forms an important basis for the reuse of components at the architectural level.

k) It forms a common basis for the communication of the various stakeholders of a system. Since these persons very often do not share a common background or methodology, interdisciplinary decision making is a key issue.

1.3 Modeling the System Behavior

As already mentioned, an architecture must be an integral part of the development methodology of an ICT-system. Such a development methodology should meet the following *requirements* [10]:

1. It should be based on uniform paradigms that can be used for the whole design trajectory from the modeling of the process to be supported by the system down to the implementation.
2. Equal emphasis should be given to the modeling of the static and the dynamic structure of the system, i.e. the modeling of the components and their interfaces as well as the behavior.
3. It should support the specification and verification of non-functional constraints during all development phases.

A visualization of the above requirements is shown in Figure 1: it gives an overall view of the five most important development steps of an ICT-system together with the four most important architectural dimensions along

which the system must be refined. The *development steps* (not necessarily of the project management phases) shown in Figure 1 are:

a) the modeling of the process to be supported[1];
b) the design of an architecture of the system that has to support the process;
c) the design; and finally
d) the implementation of the system.

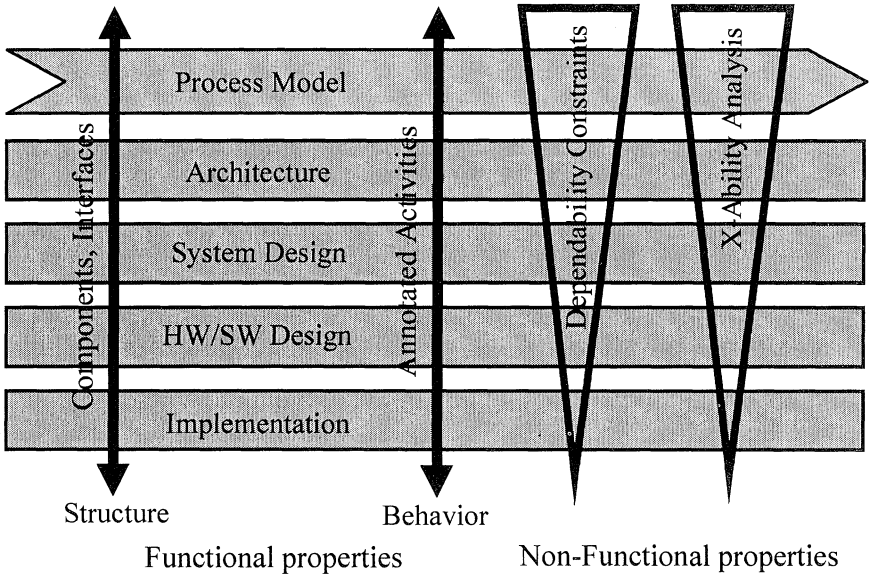


Figure 1: Uniform development paradigms for ICT-systems

The four most important *design dimensions* shown in Figure 1 are:

a) The functional dimensions structure and behavior. The two double-arrow lines indicate the (de)composition of the system in terms of components and activities (see paradigms 1.3.1 and 1.3.2 below).
b) The non-functional dimensions dependability and other X-abilities. The two triangles indicate the refinement of the non-functional part of the specification (see paradigm 1.3.3 below). In addition, they indicate that

[1] Note that this process can either be a physical production process, an information providing process, a logistic process, a service process or the process of using a particular system.

the verification of the non-functional system constraints should be done at various levels of refinement (see paradigm 1.3.4 below). Ideally, the specification and verification of the non-functional properties should be performed simultaneously with the refinement of the functional ones, i.e. the feasibility of the dependability and X-ability constraints should already be checked during the early project phases and especially in the architecting phase. The width of the triangles indicates the inaccuracy of the analysis that decreases as more implementation details are known and the estimates become better.

In order to fulfill requirements 1 to 3, four uniform development *paradigms* (i.e. paradigms that can be used at all levels of refinement) are proposed:

1.3.1 Component-based Modeling of System Structure

Components are usually application-oriented entities that usually come in form of a set of modules or even complete frameworks. Szyperski defines a component as a unit of composition with contractually specified interfaces and explicit context dependencies only [31]. Although he states that "context dependencies are specified by stating the required interfaces and the acceptable execution platform(s)", he only considers the functionality of components and does not explain what "acceptable execution platform" means. In order to be useful for resource constraint software architectures, this definition must be made more explicit. The purpose of this chapter is to show how this can be done in practice.

Along the lines of Szyperski, components are considered as independent units of deployment level, i.e. as binaries. The reason is that for source code components a much larger environment must be specified: not only the execution platform but also all code transformation tools (compiler, linker, etc.), including their switches. This added complexity also implies that the executables, resulting from the use of different sets of transformation tools, should undergo an additional regression test.

1.3.2 Activities/Transactions for the Modeling of the System Behavior

In order to meet requirement 2, the component-based paradigm needs to be extended: to model the system behavior and to support dependability analysis, the notion of an *activity* is introduced. An activity is a sequence of causally related component interactions. It models all relevant execution paths at a particular level of abstraction by a directed graph. In this graph,

the vertices are component operations (e.g. method calls) and the edges are precedence relations.

Since the objects of a distributed real-time system are assigned to different processors, activities may proceed from one processor to another via synchronous or asynchronous interactions (e.g. remote procedure calls) and have a system-wide scope. Activities are the units of concurrency and several activities may coexist in a system. Activities may communicate and synchronize via shared components like semaphores and shared data. Note that asynchronous component interaction creates additional parallelism because the called or newly created activity executes in parallel with the calling one.

Transactions are then (part of) activities for which particular consistency constraints hold. An important type of consistency constraints is atomicity. *Atomicity* typically comes in different forms like concurrency atomicity (serializability) or failure atomicity (if a failure occurs, the transaction is either executed correctly or not at all).

The concept of activities is similar to use cases that were first advocated by Jacobson [14]. An extension of use cases, called use case maps, has been described by Buhr [5]. An important difference between data flow models, message sequence charts ([12] and [13]) and similar methods, that essentially model interconnections and interactions between modules, is that use case maps concentrate on the end-to-end behavior across several modules. However, the drawbacks of use case maps are that this approach is informal and does not support the specification of non-functional constraints as described below. The activity concept thus tries to unify the above mentioned notions.

1.3.3 Specification of Dependability and X-ability Constraints by Means of Annotated Activities

In order to meet requirement 3, the next two paradigms are proposed.

Non-functional constraints must be defined end-to-end, i.e. between the stimulus from and the response to the environment. These end-to-end constraints can be conveniently specified as activity annotations, as described in [10]. A similar extension of message interaction diagrams with timing constraints is described in [6].

The introduction of additional constraints that do not follow from the system requirements should be avoided because they restrict the freedom of the designer in an unnecessary way. Nevertheless, many approaches try to decompose non-functional constraints in parallel with the componentization of the system. A typical example is the decomposition of timing constraints

that introduces unnecessary complexity and easily leads to infeasible schedules.

Non-functional constraints must also be defined in a platform-independent way in order to support portability and reusability. In this manner, there is clear separation between the requirements and their implementation in terms of a particular execution platform.

1.3.4 Verification of the Dependability and X-ability Constraints Simultaneously with the Refinement of the Static and Dynamic Structure of the System

For the verification of the non-functional constraints, many methods are available. The annotated component interaction diagrams together with the component specifications (see subsection 2.2) can be used for the generation of the necessary input. The related analysis models are summarized in subsection 2.3.5. Without going into detail, the analysis methods can be categorized as shown in Table 1:

Table 1: Verification methods for different dependability aspects

Dependability aspect	**Verification Method**
Timeliness	Schedulability analysis
Performance	Stochastic analysis: Queuing models
	Markov models
Reliability	Reliability analysis
Safety	Formal specification and verification
Security	Security analysis

The modeling of the system behavior by means of annotated activities has a number of advantages that are summarized below[2]:

a) Activities are an intuitive means for the modeling of the dynamic structure, i.e. the behavior of a system. Of course, the behavior is completely specified by the implementation of the methods of the various objects. In a large system, however, the overview is quickly lost because of the low level of abstraction. This is especially important for complex distributed real-time systems that consists of hundreds of objects distributed over many processors.

[2] Note that these advantages do not depend on the fact that the system structure is modelled in a component-based way

b) Activities can be decomposed simultaneously with the class structure. This is not surprising since they are just another representation of the implementation of the various classes.
c) Activities are also a convenient means for the modeling of synchronization and communication. This stems from the fact that communication and synchronization restrict the dynamic behavior of a system, i.e. the possible interleavings between different concurrent activities. By looking only at the active objects (parallel processes), it is hard to keep track of the various interactions.
d) Activities are the preferred entities for the specification of dependability constraints and X-ability features by means of annotations. The reason is that the implementation of non-functional constraints depends on the availability of sufficient runtime resources and thus is a dynamic feature. Putting dependability constraints where they belong also adds to the comprehensibility and reusability of the design.
e) The linking of dependability and X-ability constraints with activities avoids the real-time anomalies described by Bergmans and Akşit [2]. These anomalies occur if the code of a class is "overloaded" with non-functional aspects and are caused by the intermingling of orthogonal design dimensions: functionality at one hand and various dependability and X-ability aspects at the other hand. In fact, also the composition filters proposed in [2] avoid synchronization and real-time anomalies by defining these properties not as class attributes but as attributes of filters that manipulate the message communication, i.e. the dynamic properties of the system.
f) The consequent separation of different design dimensions increases the reusability of modules and classes. A typical example is the use of the same object by several activities with different non-functional constraints, e.g. for different operation modes or priorities. Attributing the classes or objects with non-functional requirements would either result in non-reusable components or in complex parameterized specifications.
g) A system model that is given in terms of components and activities can be easily translated into a scheduling model. Obviously, for this purpose the activities must be extended to describe the complete execution graph. Such a scheduling model is e.g. described in [32]. Each method invocation is modeled as a non-preemptive piece of an activity (typically a method invocation), called a bead. The resource requirements of the beads can be deduced from the implementation of the various classes. At a higher level of abstraction, these numbers would be based on estimates; at the program level, a code analyzer or execution profiler can be used.

h) A scheduling model as mentioned in the previous point, can also be used for a schedulability analysis at higher levels of abstraction. In this way, the timeliness of the system can be checked at different levels of refinement. An algorithm for the stepwise refinement of schedules is e.g. described in [7]. Since this algorithm makes use of the scheduling information of the upper level and because the typical expansion factors are small (in the order of 5 to 10), the algorithm achieves reasonable execution times despite the NP-hardness of the problem. Similar arguments hold for a dependability analysis along the other dependability dimensions.
i) Similar to use cases and use case maps, activities also help for the generation of test cases.

In a recent paper, also Ren et al.[27] notice that the functional domain of a design should be clearly separated from the timing domain. They also argue that it is necessary to specify the timing behavior of collections of modules. In their actor-based model, they define *RTsynchronizers* as independent modules that specify timing constraints on different event patterns of a group of actors. A RTsynchronizer thus restricts the temporal behavior of such an actor group. Since events are defined as the invocation of a message at an actor, this method has some similarities with the composition filters described in [2] and discussed above.

Although this approach acknowledges the separation of static from dynamic properties and the necessity to define end-to-end timing constraints, it has a number of disadvantages compared to the activity-based method advocated in this chapter. First of all, event patterns are nice abstractions of the system behavior but do not provide an intuitive view of the flow of control. In addition, there is no obvious way to decompose RTsynchronizers simultaneously with the class structure. Together with the fact that the actor groups of RTsynchronizers can overlap, the aforementioned properties of RTsynchronizers make it difficult to comprehend the timing behavior of complex systems. Finally, event patterns cannot be straightforwardly translated into a scheduling model since they do not specify the precedence relations.

1.4 What is the Design Space of Architecture?

Since separation of concerns is important for ICT-architectures, it is meaningful to separate the design space into functional and non-functional dimensions. An example of important design dimensions is given in Figure 2.

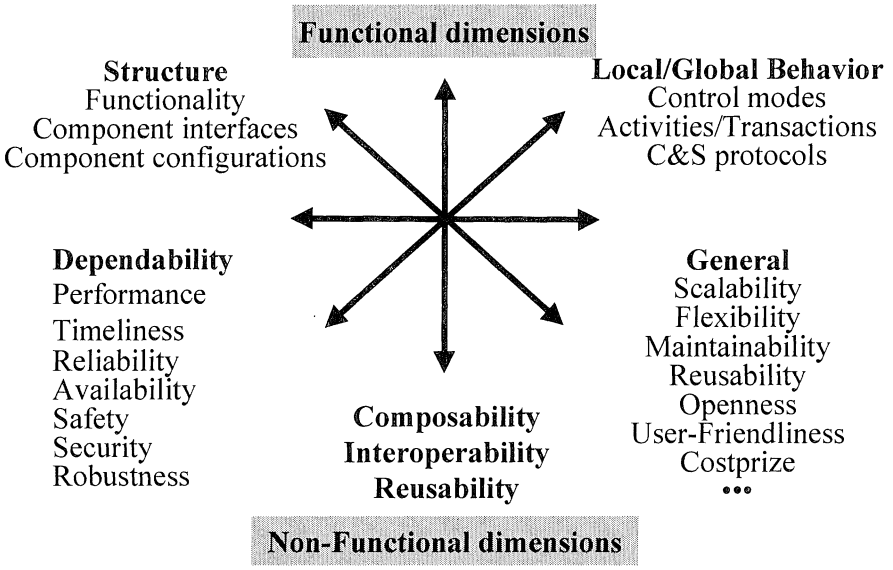


Figure 2: Important design dimensions

1.4.1 Functional Dimensions

1.4.1.1 Structure

A good architectural model should define the static structure of a system in terms of the major components, including their specification (functionality), their interfaces, their interconnections and the possible restrictions of the configuration of components and interconnections.

Although implied by a good model, the following aspects should be made explicit since they play such a vital role in the development, production and maintenance process of an ICT-system:

a) The functionality.
b) The interfaces between the various components.
c) The available configurations. These configurations are important for the sales department, the service department and the users/customers of a system.

d) The isolation of stable, changing and critical technologies into appropriate building blocks. In this way one can try to keep the influence of the often rapidly changing technology as local as possible.

1.4.1.2 Behavior

A good architectural model should also define the dynamic structure of a system. Ideally (see subsection 1.3) this is done in terms of concurrent activities/transactions, that describe also the restrictions on the possible execution paths. In practice, the possible interaction sequences between components are often specified in terms of protocols.

1.4.2 Non-functional Dimensions

1.4.2.1 Dependability

As mentioned before, this dimension is especially important for embedded and safety-critical systems. As mentioned in the beginning of subsection 1.3, dependability includes performance, timeliness, reliability, availability, safety, security and robustness.

The performance of a system is often expressed in terms of average response time, throughput (processing or communication bandwidth) and sojourn times. For embedded and safety critical systems it is usually also necessary to specify timing constraints in terms of periods, synchronization intervals and deadlines (maximal response times). Note that the specification of timing constraints is only meaningful if also the maximum load of a system is specified in terms of a load hypothesis. Reliability, availability, safety, security should be defined in terms of a hypothesis of the respective adverse events. Finally, robustness is the ability of a system to cope with unexpected adverse events that are not covered by a separate hypothesis.

1.4.2.2 General Aspects

This category is used as a collection of generally desirable features of an architecture that, depending on their importance for a particular architecture, may or may not be regarded as separate dimensions. Among the most common features are:

a) the scalability of a system,
b) the flexibility to make different configurations and variants,
c) the extendibility and maintainability of a system,

d) the reusability of building blocks of systems and system families,
e) the portability to different execution environments,
f) the openness and interoperability of a system and
g) the robustness with respect to changing requirements.

1.5 Which Views Should be Supported by Architecture?

A view describes a system with respect to some set of attributes or concerns [26]. The views to be supported by an architecture can be derived from the concerns of the various stakeholders like development-, production and service engineers, development-, product- and production managers, sales people and users/customers. The relation between the dimensions of an architecture and the concerns of the various stakeholders is shown in Table 2.

Table 2: The relation between the dimensions of architecture and typical stakeholders

	Engineering Devmnt	Product'n	Service	**Custo-mer**	**Sales**	**Users**
Structure						
Functionality	x	x	x	x	x	x
Interfaces	x	x	x			
Configurations	x	x	x	x	x	
Behavior	x		x	x		
Dependability	x		x	x	x	x
X-abilities	x		x	x	x	x

A non-exhaustive overview of the concerns of different stakeholders is shown in table 2:

a) Development engineers: For all types of development engineers (like hardware-, software- and system engineers), the architecture is the most important means for discussing alternatives, achieving consistency and enabling concurrent engineering. For complex systems, it is necessary that each discipline derives its own architectural view from the overall architecture.
b) Production engineers: Since assembly can be considered as the time-sequential establishment of system interfaces [26], production and assembly is an important view onto an architecture. This also holds for ICT-systems and especially for component-based systems. In particular, an architecture should support the design of the system production (generation) process. It is often the case that development engineers pay

too little attention to the production and assembly of the different configurations of a system. A typical problem is that they concentrate too much on the complete system and do not consider the efficiency of the assembly of different subassemblies and configurations.

c) Service engineers: For the service of a system not only the various system configurations, their functionality and their interfaces are important, but also their failure and repair characteristics. The latter are often specified in terms of (a) units of failure (the smallest units to which failures are traced down); (b) units of replacement (the smallest units the service engineer will replace); units of repair (the smallest units the service workshop will repair); and (c) units of recovery (the smallest units that can be independently recovered after repair). If appropriate, these categories should, of course, also be indicated at the architectural level.

d) Customer: Customers might be external persons or internal management. They are primarily interested in the business performance indicators like customer satisfaction, quality, time to market and costs. Since these indicators must be known as early as possible, the architecture plays an important role in their estimation. In addition, customers are also interested in general product features like configurability, dependability and X-abilities like scalability, maintainability and interoperability.

e) Sales: For the sales department, mainly the available configurations and their specifications (functionality, behavior, dependability and other X-abilities) are important.

f) Users: The users are primarily concerned about the usability and reliability of the system for a given purpose. Depending on the type of user, he might also be interested in a number of general X-abilities.

2. COMPOSABILITY CRITERIA FOR RESOURCE CONSTRAINT COMPONENT ARCHITECTURES

2.1 Introduction

Component-based software engineering and component-based architectures become increasingly important because of their potential to increase the software productivity by supporting reuse. The availability of modern middleware in the form of ORB's (Object Request Brokers) enables the construction of distributed component-based systems in a transparent way, i.e. independent from the distributed nature of the system and the

underlying communication network. Although components do not need to be objects, they share many concepts and software engineering advantages with object-oriented approaches. Important features of components are modularity, encapsulation (i.e. a clear separation between specification/interface and implementation) and interface inheritance. Note that nothing is told about the implementation of components: it is perfectly possible that they are constructed in an object-oriented way, including implementation inheritance.

One of the greatest challenges of component-based architecting is to make architectures compositional, i.e. to ensure that all envisaged combinations of architectural building blocks (e.g. the components of a product family) work together without problems. A more exact definition is, that an architecture is composable if a property that was established at the component level, will not be invalidated if it must cooperate with other components [18]. Important examples of such properties are timeliness, reliability, safety but also scalability and testability. Note that this definition is less stringent than the one usually used in the context of formal methods. There, a formal model and its proof system are said to be composable, if the correctness of a complete system can be established by using only the (independently proven) features of its components.

Composability is especially important for resource constraint systems. Even if the functional interfaces are correct, problems tend to arise in many other areas that are related to the use of shared resources that are scarce. Examples are missed deadlines due to race conditions and schedulability problems, reduced reliability due to unexpected component interference, slow-down or blocking due to constraints of the memory or communication system and many other "side effects".

Component composition should be treated in the context of an architecture because this allows the specification of important global system properties like interaction styles and constraints. The basic ingredients of any architecture are components, interfaces and the relations between components. The latter can be static (common to all runs like communication channels and shared data structures) or dynamic (occurring in a particular run like message interaction and synchronization). With respect to defining the relations between components and thus composability, current architecting practices for real-time systems have a number of significant shortcomings. Among the most important are:

a) Conventional descriptions of architectures concentrate too much on the structure of the system and do not adequately deal with the dynamic aspects. The emphasis is on the componentization of the system and the description of the component interfaces, but not on the precise

description of the behavior. As systems become more complex, it is, however, important to restrict the behavior to certain well-defined and understood modes.

b) Interface descriptions usually focus on the functional part and tend to neglect resource requirements and interaction styles. Even if additional aspects of the various interaction channels like protocols are specified as part of the interface, there are no restrictions on the actual interaction patterns and behavior that may occur at runtime. This is especially true if the interaction patterns in the time-domain are not specified. Consequently, there is no guarantee that these components are composable.

c) There are no good approaches for tackling the non-functional requirements, like dependability (performance, timeliness, reliability, availability, safety, security and robustness), scalability, flexibility, usability, maintainability and reusability. These system aspects should be treated as independent design dimensions, specified in an implementation-independent way and implemented in such a way that the resulting behavior becomes predictable. Non-functional requirements should be stated in the requirements and taken into consideration from the beginning, i.e. already during the architecting phase. Unfortunately, current software engineering practice often relies on Moore's law (the hope that the abundant availability of computing power will decrease the chance for conflicts related to computing and memory resources) and considers non-functional requirements only during the implementation phase. Obviously, the explicit treatment of the non-functional requirements from the very beginning of the development is a prerequisite for composability.

The non-functional properties of a system can be considered as a special type of 'emergent properties' [28], because they are related to the overall behavior of the system and depend on the coordinated action of a great number of, or even all, components. They can thus not be tackled in a simple reductionistic way, i.e. by designing these properties only into individual components.

A good component architecture can support emergent properties by defining principles for their uniform implementation. These principles are described in terms of appropriate architectural styles, distributed algorithms and protocols that govern the component interaction. They can be either implemented at the application level (i.e. in all relevant components) or as generic services in the execution platform. The latter is preferable because the emergent properties can be effectively implemented in a uniform and reliable way. Typical generic services for distributed real-time systems are

clock synchronization, scheduling, membership, atomic broadcast and support of replica determinism. Note that there is no fundamental difference between component architectures and component frameworks since also frameworks often implement particular component interaction styles and algorithms.

An example of an emergent non-functional property is the enhancement of the reliability by component replication. For this purpose, multiple copies of a component are installed, preferably on different processors of a distributed system. These copies get the same input, execute in parallel and produce a single output via a voting algorithm. In order to keep the replication transparent at the application level it is convenient to have operating system services that facilitate the instantiation of multiple instances of a component, the distribution of the input and the voting of the output. In addition, the operating system should support replica determinism by ensuring a distributed schedule that guarantees the simultaneous execution of the code of the replicated components, including the consumption of the inputs and the generation of the outputs.

The correct and consistent implementation of emergent properties is essential for achieving composability. At the moment, the composition problem can only be solved for architectures that are based on restrictive, and often unrealistic, assumptions. The most prominent example are Time-Triggered Architectures (TTA's) (see [17] and [18]) that are based on a strictly deterministic environment (100% assumption coverage) and mainly suited for periodic hard real-time systems. For the much larger class of Event-Triggered Architectures (ETA's) there are, however, no good general concepts for dealing with composability. This is mainly caused by the fact that the time behavior of ETA's is not strictly controlled in order to achieve flexibility and high resource utilization.

In ETA's, the correct composition of components has many functional aspects (like the specification of functions and invariants) and non-functional aspects (like timeliness and reliability). In order to arrive at a compositional architecture, the following conditions must be met:

a) The specification of components must cover all aspects that influence the interaction with other components. This requirement comprises two main categories:
   - *Resource usage*: It is necessary to specify not only the functions provided by a component but also all resources it uses for this purpose.
   - *Architectural styles* used or assumed in the interaction with the environment: Architectural mismatches are very difficult to work-around and are responsible for many problems with the integration of

components, that are independently developed or belong to different component frameworks, into a given architecture.

b) All non-functional constraints must be defined separately from the functional specifications of the components, e.g. by means of component interaction diagrams. The reasons are:

   - These constraints are emergent features and depend not on individual components but also on their interaction and on the available resources. The fulfillment of the non-functional constraints is thus a dynamic feature.
   - Components become more reusable. This holds especially for the time domain that is heavily dependent on the execution platform (e.g. clock rate and available memory). It is for example not appropriate to specify timing constraints (deadlines, periods, etc.) at the component interface because the component may be used in different real-time contexts.

2.2 Interfaces

In order to achieve composability, the specification of a component must include all aspects that influence the interaction with other components in any way. Special attention must be paid to active components that have one or more concurrent threads. This requires for each component the specification of the following aspects:

a) *Functionality* of the component services:
   In order to specify the functional part of a component interface, the algebraic style that is used for object-oriented specifications is well suited.
   - Operations: Pre- and postconditions
   - Attributes: Preferably, attributes should be read-only and changeable only via the operations. Note that stateless components are much easier to compose because their behavior does not depend on previous interactions, i.e. on their history.
   - Exceptions: All exceptional states that the component can not handle itself.
   - Invariants: All conditions that must hold for the whole component at the beginning and at the end of an operation. Also the constraints on the concurrency of the various operations (sequential or concurrent execution) can be expressed by means of invariants. Concurrent execution means either that the component serializes request

automatically or that requests can be arbitrarily interleaved. In the latter case, reentrancy is an important property.

b) Estimates about the *resource requirements*:
   - Execution per operation and thread: Minimum, average (distribution) and maximum execution time. Often only the Worst-Case Execution Time (WCET) is specified. Note that this metric is platform dependent.
   - Memory: Minimum, average (distribution) and maximum required primary of any type. For this purpose, a platform independent metrics (e.g. bytes or 32-bit words) can be used.
   - Fault hypothesis: All (combinations of) faults the component is supposed to tolerate.
   - Software reliability in a general metric like estimated faults per kByte code.
   - Threat hypothesis: All (combinations of) security threats the component is supposed to tolerate.
   - Dependencies: If not all operations and threads have the same dependencies, these must be given at a lower granularity than a component because not all operations or threads might be used in every application. They include services used from other components, including hardware components, system services (operating system, communication system, ORB, DBMS, etc.) and libraries. In order to be able to handle hardware resources as components, wrappers are usually used for their encapsulation. Typical hardware resources are sensors, actuators and peripherals. Whether they can be used concurrently or not (preemptive or non-preemptive) can be specified by appropriate invariants (see point a). Note that the dependency relation is transitive.

c) *Architectural styles* used or assumed in the interaction with the environment. There must be a match between the architectural styles used within a component and the architectural styles used for the overall system. For a detailed description of these problems associated with architectural mismatches see [3] and [9]. In order to describe the interaction styles, first a *Concurrency model* must be defined that gives the number of concurrent threads of the component: 0 for a passive component, 1 for a single-threaded component and n for a multi-threaded component. For active components, also the following models must be known:
   - *Trigger model*: Time-triggered or event-triggered. In a time-triggered model, an active component is driven by its internal clock (the local clock of the execution platform) and polls its environment at regular

intervals. In an event-triggered model, its environment drives an active component, e.g. by events or operation calls.

- Interaction model in terms of:
  * Topology: Geometric form of the control- or data-flow.
  * Multiplicity: Point-to-point, multicast / broadcast; directed or undirected communication.
  * Sharing: Centralized or distributed; communication via a shared blackboard or messages.
  * Initiative: Publisher/subscriber or client/server; implicit activation of the operations by the server or explicit invocation via the client. Note that the client/server style is meant here in the object-oriented sense and not in the networked sense.
  * Periodicity: Continuous, periodic or aperiodic; the interaction is continuous, periodic or aperiodic. Continuous interaction occurs e.g. in hybrid systems or in pipe-and-filter architectures.
  * Synchronicity: Synchronous, asynchronous or datagram; in the first two cases, the caller waits on the results (RPC) or continues in parallel and picks-up the result later; in the last case, a single message is exchanged.
  * Autonomy: Inactive, active or mobile entities; passive stationary components, active stationary components or mobile autonomous agents.
  * Protocol: sequence of related interactions. Often different modes can be distinguished like initialization/set-up, normal operation and closing.
  * Binding: Compile-, instantiation-, initialization- or invocation-time; defines when is the identity of the interaction partners is established.

  Note that the synchronicity of the interaction (synchronous, RPC or asynchronous) is implied by the signature of the various operations.

d) *Reflection interface* (similar to the reflection API in Java) that allows to infer the properties of a component at design-time or at run-time. The latter is e.g. necessary if components are dynamically instantiated by an ORB. Such a reflection interface must allow to infer at least the following information:
   - General description of the features and purpose of a component.
   - Complete description of the interface as explained above.
   - Known problems.

For the description of these interface attributes, it is important to keep Meyer's contract principle [22] in mind: If the client of a service obeys the preconditions, the component must guarantee the postconditions. Even if a

component is not specified in a formal way, it is worth while to document the pre- and postconditions informally. It is important to realize that this holds mutatis mutandis also for the architectural styles.

It should not be too difficult to extend the CORBA (Common Object Request Broker Architecture) IDL (Interface Definition Language) for the specification of the above mentioned features.

2.3 A Design Method for Distributed Real-time Systems

In general, the methods that were developed for the design of object-oriented systems can also be used for component-based systems. An exception is the use of inheritance: since inheritance breaks encapsulation, there is still much debate about its use in component-based systems.

One great advantage of the object-oriented approach is that there is a clear separation between the structure of the system (described by class-, object-, package-, component- and deployment diagrams) and its behavior (described by use case-, interaction-, activity and statechart diagrams). Thereby, statecharts are often used for the description of the local behavior of components and different types of interaction diagrams for the description of the emergent behavior of a collection of components. This allows also a strict separation between the specification of functional and non-functional features. As described in subsection 1.3, the latter are dynamic features and related to the interaction of components. Another advantage of this object-based approach is that it can be used at all levels of abstraction from the system architecture down to the implementation. In this way, the use of multiple incompatible modeling techniques (e.g. entity-relationship diagrams, data-flow models, finite-state machines, etc.) and the resulting interpretation and translation faults can be avoided.

For the modeling of distributed real-time systems a number of specialized methods have been developed like Real-Time UML (Unified Modeling Language) [6] and MSC (Message Sequence Charts) ([12] and [13]) that both include (message) sequence diagrams that can be annotated with timing constraints. In subsection 1.3, activities were proposed as a concept that unifies the above mentioned approaches and extends them with respect to the other dependability constraints.

The design of complex systems requires several levels of abstraction that are usually designed in an intermingled fashion with much iteration in between. In addition, the top-down refinement is usually combined with bottom-up composition of components. At each level of abstraction, the components (including their constituent classes and objects) are refined together with the activities that describe their interaction. In the approach for the construction of dependable distributed real-time systems that is proposed

below, the design at each level of abstraction is separated into six steps. Although these steps are listed in logical order, they are usually intermingled with much iteration in between. These six steps are:

2.3.1 Construction of a *Process Model*: What must be supported or controlled?

Systems and their architectures should be derived from the process that the system is intended to support. This process can be a production process (continuous or discrete fabrication, workflows, etc.) or a usage process (describing the use of the product or system by different classes of people like customers, users, developers, maintainers and service personnel). A big advantage of this approach is that the development is automatically focussed on the behavior and the usability of the system for the stakeholders.

This model describes the essential classes, objects and activities in the environment that must be supported or controlled by the system. Examples of classes and objects are sensors and actuators, user-interface devices, ICT-devices and additional entities necessary to model the behavior of the environment. Their specification includes functional constraints like pre/post conditions and invariants.

Up to now, the development of embedded systems was mainly driven by technology. This was quite natural in a technical environment where the system usually interacts with other systems and the users are either other technicians or quite distant and abstract for the developers. In today's competitive business environment, however, this attitude is not adequate anymore because developments are increasingly driven by the primary business. Examples are discrete or continuous production systems, that are more and more driven by logistic systems that control a business process that is carefully (re)designed to fulfill the customer requirements. Other examples are consumer products that are less and less bought because of their technical merits but because of their ability to support a certain user activity like the collection of information, the support of a personal job or entertainment. A third class of examples is all sorts of command and control systems that are used in cars, planes and military systems. Also here, looking at the processes to be supported can often considerably reduce the complexity of the system and especially of the user interface. In all cases, the end-users enter the picture and the Embedded System must be designed to support a user- or application process.

2.3.2 Construction of a *System Model*: How must the supporting or controlling system be constructed?

The components and activities of the supporting or controlling system are derived from the process model. If necessary, supporting classes, objects and activities are derived from the controlling ones.

2.3.3 Construction of a *Distribution Model*: How must the various components be distributed and connected?

This model describes the distribution of the components over the available resources. In order to support communication, it might be necessary to add communication components and to extend the activities. The communication times can be modeled by introducing network components that introduce the appropriate delays. Alternatively, the communication delays can be directly introduced in the scheduling model (see subsection 2.3.5).

The explicit modeling of the distribution of the software over the processors of a distributed system is important for some applications and transparent for others. In the latter case, a (semi-)automatic scheduler can first solve the assignment problem and then construct a local schedule per processor.

2.3.4 Construction of a *Concurrency Model*: Which components and objects are active or passive and what dependencies (synchronization and communication via messages or shared resources) exist between the different activities?

This model is usually implied by the architectural styles used in the system. It describes the active objects that start, and possibly also terminate, activities. For these active objects, the following additional models must be defined as described in subsection 2.2:

a) *Trigger model*: Time-triggered or event-triggered.
b) *Interaction model* in terms of topology, multiplicity, sharing, initiative, periodicity, synchronicity, autonomy, protocol and binding.
c) Note that the synchronicity of the interaction (synchronous, RPC or asynchronous) is implied by the signature of the various component operations.

2.3.5 Construction of a *Dependability Model*: How can dependability be achieved in a predictable way?

For the verification of the non-functional constraints, many methods are available. In addition, a worst-case stress-analysis is important for all dependability aspects. The annotated component interaction diagrams together with the component specifications can be used for the generation of the necessary input. For the different dependability aspects, the following models must be constructed in order to apply the appropriate analysis methods and techniques (see also table 1):

a) Scheduling model: Schedulability analysis based resource usage of components, activity activation frequencies and deadlines.
b) Reliability model: Reliability analysis based on reliability figures of hardware and software objects, activity activation frequencies and reliability requirements.
c) Availability model: Analysis based on repair times of hardware and software objects, activity activation frequencies and availability requirements.
d) Safety model: Analysis of the absence of catastrophic states.
e) Security model: Security analysis based on threat probabilities of objects, activity activation frequencies and security requirements.

2.3.6 *Evaluation & Improvement*: How can the resulting system be improved?

The improvement must be based on a careful evaluation of all functional and non-functional features of the system. The introduction of more resources or the reduction of the functionality should only be considered if strictly necessary. In principle, the following measures can be taken:

a) Efficiency enhancement by clustering of objects and components:
   - Clustering of active entities based on temporal-, control- or functional coherence.
   - Migration of passive entities used by more than one activity and whose resource space shows considerable overlap with active objects.
   - Clustering of common passive entities into larger components.
b) Performance enhancement by introduction of more concurrency:
   - Reduction of blocking times.
   - Splitting of critical activities into concurrent ones.
c) Dependability enhancement by introduction of more resources or reduction of the functionality.

2.4 Example

The approach described in subsection 2.2 and 2.3 is demonstrated by means of a simplified example of a Car Navigation System (CNS). Such systems are aimed at assisting the driver of a car to efficiently navigate through an area. To this end, the system has a display that shows a map of the area around the location where the car is driving. A special symbol on the map indicates the current position of the car. The map and the location of the car-symbol are updated as the car moves around. The display is located together with a keypad on a console in the dashboard of the car. The driver can use the keypad to enter commands. Typical commands are: "display a particular map", "zoom in/out" and "find a route from where I am now, to a particular destination". If a route has been requested, this route is shown on the map of the display.

This case study describes the architecture of the car navigation system. To this end, the UML notation and process as described in [4] is used. The Rational Unified Process [20] uses the 4+1 view model initially suggest by Kruchten [19] for describing an architecture from different viewpoints. These five views are

a) the *use case view* describing the behavior of the system as seen by the stakeholders in terms of use case models;
b) the *design view* comprising class diagrams, object diagrams, interaction- or activity diagrams and statechart diagrams;
c) the *deployment view* describing the mapping of the software components onto the hardware components in terms of component diagrams (software topology), deployment diagrams (hardware topology) and possibly also interaction- or activity diagrams and statechart diagrams;
d) the *process view* that is similar to the design view but emphasizes the active classes and objects (processes and threads);
e) and finally the *implementation view* encompassing the software entities (files) used to assemble the system in terms of component diagrams and possibly also interaction- or activity diagrams and statechart diagrams.

The 4+1 view model can be mapped onto the first five design steps for distributed real-time systems described in subsection 2.3 as follows:
a) The use case view can be considered as a greatly simplified process model.
b) The design view corresponds to the functional part of the system model. Note that neither the use case view nor the design view supports the specification of non-functional requirements like dependability.

c) The deployment view corresponds to the distribution model of the proposed method.
d) The process view corresponds to a simplified version of the concurrency model of the proposed method.
e) There is no equivalent to the implementation view since software configuration management is not of concern here.

For this case study, the most interesting views are the use case view, the design view and the deployment view. For the design view, a class diagram and a sequence diagram will be presented. An extra object diagram is not needed since every class of the CNS system is instantiated only once when the system is initialized. Also a separate process view is not used since the active classes and their interaction can be indicated in the class diagram. In order to simplify the presentation, the deployment diagram subsumes also the component diagram.

2.4.1 Process Model

As described in subsection 2.3, the method starts with a process model. For this example, a use cases model is used.

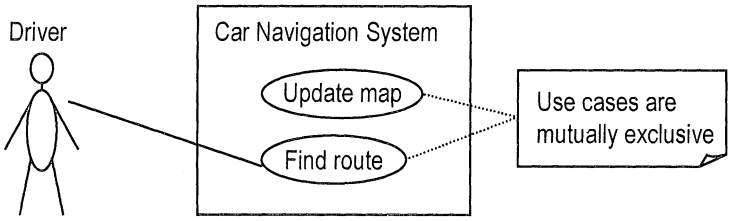


Figure 3: The use case diagram

As the car moves, the CNS system automatically updates the map on de display and the position of the car ('Update map' use case). The driver of a car may also instruct the CNS to find a route ('Find route' use case). To this end, the driver must enter a destination via the keyboard, possibly after having requested another map. After the CNS has calculated a route, the map is updated and the positions of the car and the destination are indicated. These use cases are shown in Figure 3.

For this example, only 'Find route' is considered, which is the more complex of the two use cases. In order to simplify the schedulability analysis (see below), it is assumed that the two use cases are mutually exclusive. This is not a big restriction because also the 'find route' use case comprises an

update of the display; only the writing of a new map into the display buffer (the 'Blackboard') may take more time because of the calculations involved.

2.4.2 System Model

The next view that is described is the design view, which coincides with the system model. This view will be described in terms of a class diagram (Figure 4) and a sequence diagram (Figure 5).

First, the classes that constitute the system are briefly described. The CNS contains a global positioning subsystem 'GPS' to determine the position of the car. The 'GPS' system consists of a 'Receiver' (GPS functions through reception of several satellite signals) and a processing unit, called 'Position', which computes the location from the receivers' measurements.

Furthermore, the CNS contains a 'RouteDB' subsystem, which is responsible for generating maps and computing routes for given source and destination locations. To this end, it consists of a route 'Database', a 'Query Manager' that manages the access to the database and a 'RoutePlanner' which executes the algorithms necessary to find a route from source to destination. Such planning problems are, in most cases, NP-hard and their approximation needs abundant processing resources.

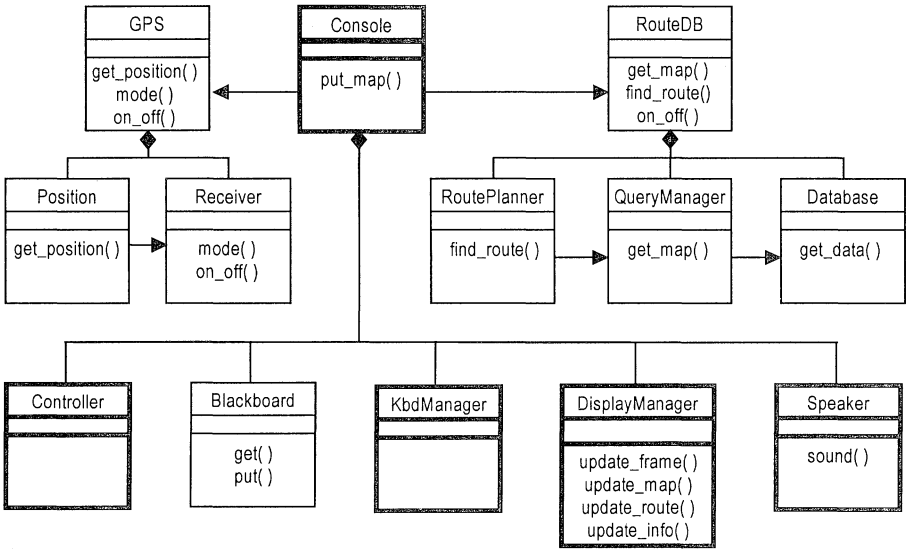


Figure 4: The class diagram

The third subsystem, the 'Console', consists of a keyboard managed by class 'KbdManager', a display managed by class 'DisplayManager', a 'Controller', a 'Blackboard' and a speaker managed by class 'Speaker'. These classes have the following responsibilities. The 'KbdManager' scans the keyboard matrix with a period of 0,1 s (this is an internal of 'KbdManager' and not shown in Figure 4), combines the characters entered until the next enter-key to a command-line and writes the command line into the 'Blackboard'. The command-line buffer of the 'Blackboard' has room for several commands. The 'Blackboard' is a shared repository for data within the 'Console'. It also functions as a video-buffer for the 'DisplayManager'. The latter polls the contents of the blackboard every 20 ms. Also the 'Speaker' polls the 'Blackboard' with a period of 100 ms for warning signals to the driver. The 'Controller' manages the cooperation of the classes within the 'Console' and also the interaction with other components. To this end, it scans the command buffer of the 'Blackboard' every 100 ms and takes the appropriate actions.

Since all classes, except the 'Blackboard', show autonomous behavior they are modeled as single-threaded active classes.

The dynamics of the system is modeled using sequence diagrams. In Figure 5, only a sequence diagram, which models a possible execution for the 'Find route' use case, is given.

The sequence diagram shows an example of a possible execution of the system. This execution starts with the controller polling the keyboard for commands. However, no command has been entered yet. Next, the Displaymanager, and subsequently, the speaker, check whether the blackboard contains any data for them. The polling periods for these respective checks are 20ms and 100 ms (indicated alongside the dotted left bracket). While these polling actions are going on, the user enters a "find route" command and a destination. This command should be retrieved by the controller from the keyboard manager within 1s after the previous time that the keyboard manager was polled.

Once the controller finds a "find route" command, it proceeds by requesting the current position from the GPS. This position is used as the source for the route to be found. Subsequently, the controller sends a "find_route(src,dstn)" request to the Route DB. This request is forwarded to the Route Planner. The route planner then requests the maps it needs from the actual Database via the Query Manager. In the execution depicted, two such requests are shown, the dotted lines between them indicate that this sequence may repeat a number of times. When the planner has computed a route, the corresponding map and route are written into the (video buffer of the) blackboard. This data is retrieved from the blackboard by the display manager and made available to the driver of the car.

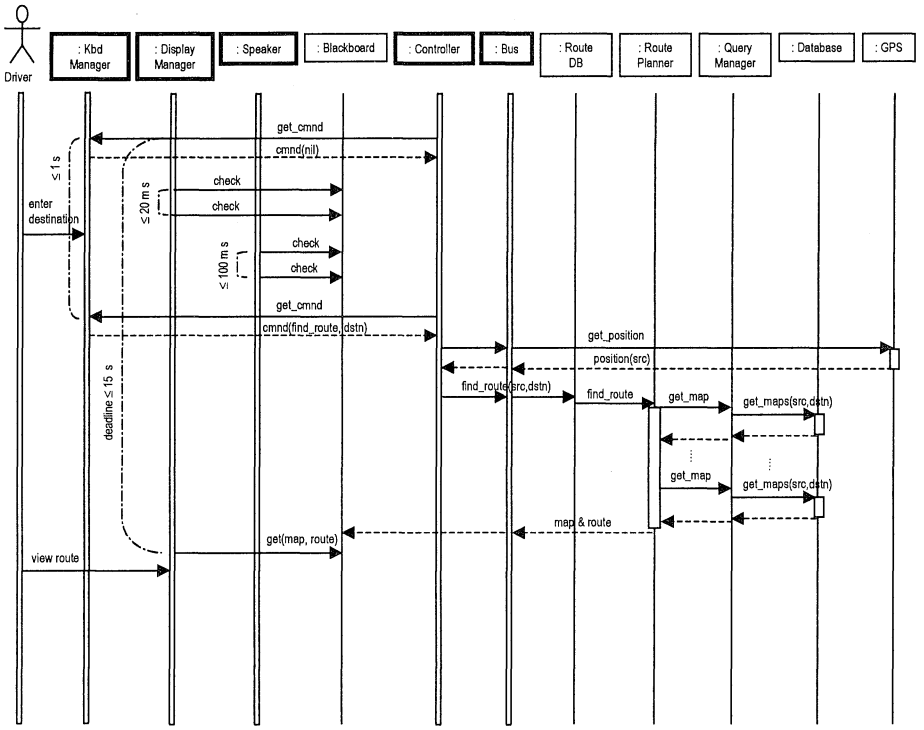


Figure 5: A sequence diagram for the 'Find route' use case

2.4.3 Distribution Model

Next step of the proposed method is the definition of a distribution model, i.e. a model that shows how the components are distributed over the nodes. This yields the deployment diagram shown in Figure 6.

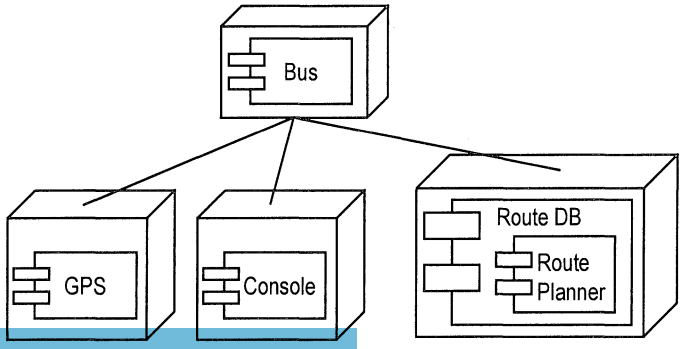


Figure 6: The deployment diagram.

Based on the class-diagram, four components are identified: a 'GPS' component, a 'Console' component, a 'RouteDB' component and a 'RoutePlanner' component. Furthermore, the communication between the components is explicitly modeled by using a 'Bus' component. Modeling the bus as a component allows us to treat its non-functional features in the same manner as all other components. This also allows us to hide the internals of the bus like the protocol stack that usually consists of a physical layer and a Medium ACcess (MAC) layer. In order to abstract from the MAC protocol it is assumed that a bus access for transmitting a single message takes at most 40 μs.

The 'Bus' and 'GPS' components are both mapped onto single nodes because these are commercially available this way. To minimize communication cost, the 'RouteDB' and the 'RoutePlanner' are mapped together onto one node. The remaining component, the 'Console', is also mapped onto a single node. All components communicate via the 'Bus', using the Client/Server paradigm.

In order to ensure the correct cooperation between the components and to facilitate the schedulability analysis, their interfaces must be described along the lines presented in subsection 2.2. To this end, the templates shown in Table 3 are used:

Table 3: Interface specifications of the components

Component name	Console
Functionality	See class diagram
Resource Usage per op.	n.a.
Worst Case Exec Time	n.a.
Max. Memory use	5 Mb
Dependencies	GPS, RouteDB and Bus
Peripherals	Display device and Keyboard device
Architectural Style	
Topology	Bus connection to GPS and RouteDB
Multiplicity	Point-to-point
Sharing	Distributed
Initiative	Client/Server
Periodicity	Aperiodic
Synchronicity	Synchronous
Autonomy	Active
Protocol	Non

Component name	GPS		
Functionality	See class diagram		
Resource Usage per op.	Get_Position	Mode	on_off
Worst Case Exec Time	1 sec	1 μs	Off: 55 s (cold start)
Max. Memory use	128 kB	n.a.	n.a.
Dependencies	Non		
Peripherals	n.a.		
Architectural Style			
Topology	Bus connection to Console		
Multiplicity	Point-to-point		
Sharing	Distributed		
Initiative	Client/Server		
Periodicity	Aperiodic		
Synchronicity	Synchronous		
Autonomy	Passive		
Protocol	Non		

Component name	RouteDB		
Functionality	See class diagram		
Resource Usage per op.	Get_map	find_route	on_off
Worst Case Exec Time	1 sec	11 sec	1 μs
Max. Memory use			n.a.
Dependencies	RoutePlanner		
Peripherals	n.a.		
Architectural Style			
Topology	Bus connection to Console, direct connection to RoutePlanner		
Multiplicity	Point-to-point		
Sharing	Distributed		
Initiative	Client/Server		
Periodicity	Aperiodic		
Synchronicity	Synchronous		
Autonomy	Passive		
Protocol	Non		

Component name	RoutePlanner
Functionality	See class diagram
Resource Usage per op.	Find_route
Worst Case Exec Time	11 sec. (assuming route DB performance of 1 sec per request)
Max. Memory use	4 Mb
Dependencies	RouteDB
Peripherals	n.a.
Architectural Style	
Topology	Direct connection to RouteDB
Multiplicity	Point-to-point
Sharing	Distributed

Initiative	Client/Server
Periodicity	Aperiodic
Synchronicity	Synchronous
Autonomy	Passive
Protocol	No

Component name	Bus
Functionality	Not considered here
Resource Usage per op.	4 Mbit / sec
Worst Case Exec Time	40 µs
Max. Memory use	n.a.
Dependencies	No
Peripherals	n.a.
Architectural Style	
Topology	n.a.
Multiplicity	Point-to-point
Sharing	n.a.
Initiative	Client/Server
Periodicity	Aperiodic
Synchronicity	Datagram
Autonomy	Passive
Protocol	n.a.

Since all components obey compatible architectural styles, no conflicts occur.

2.4.4 Dependability Model

Finally, the timing model as part of the dependability model is constructed. In order to do that, more details about the underlying hardware are needed. The display has 1024*1024 pixels each of which may one of 256 colors. Hence the size of a videomap is 1 MByte. Given a bandwidth of 4 Mbit/sec, it takes 2 s to send a videomap over the bus. The time needed for sending a position (a pair of coordinates) is negligible compared to that for sending video-maps (order of microseconds).

2.4.4.1 Deadline Analysis

To check whether the deadline of 15 s can be met, the worst case execution times of the actions that need to be performed in order to compute a route are summed-up, as shown in Table 4:

Table 4: Worst-case execution times relevant for the deadline of the 'Find route' use case

Operation	Time [s]
GPS.get_position	1
RouteDB.find_route	11
Send map over Bus	2
Blackboard.put(map)	0,02
Total	~14

2.4.4.2 Schedulability Analysis

Next we need to make sure that all components are schedulable on their respective nodes without conflicts. In the sequel, the schedulability per node is considered.

The 'GPS' component is the only component executing on his node and the WCET of operation 'GPS.get_position' allows the meeting of the deadline as shown in Table 4.

For the Route planner and the Route_DB the assumed WCET's guarantee their schedulability.

For the 'Console' component, it is assumed that the system is running on a 75 Mhz processor under a real-time operating system. Furthermore, it is assumed that the processor can read and write 1 word per cycle, hence 75.10^6 words/s. Now, several components and active objects must be scheduled. Since we work at the architectural level, rate-monotonic scheduling is assumed because of its simple schedulability condition. Note that at lower levels of abstraction (e.g. at the code level) where all use cases need to be scheduled, other scheduling paradigms (e.g. earliest-deadline-first) might be necessary. In order to simplify the schedulability analysis, it is further assumed that the deadline of the 'Find route' use case is equal to its period.

a) The 'Controller' scans the 'Blackboard' with a period of p = 100 ms (=$1,0.10^5$ µs). The total execution time is c = 250 µs. The resulting processor utilization thus amounts to c/p = $2,5.10^{-3}$.
b) The 'DisplayManager' needs to refresh the screen at 50 Hz, i.e. with a period of p = 20 ms. The transfer of 1 Mbyte of video buffer by a 75 Mhz processor takes c = 14 ms. Hence c/p = 0,7.
c) The 'KbdManager' needs to poll the keyboard and dispatch commands with a period of p=100 ms. Polls to the keyboard cost 10 µs, hence c/p = 10^{-4}.
d) For the 'Find_route' use case, the 'Controller' finally needs to write the map and the route into the 'Blackboard'. It is assumed that for the

transfer of the map one read and one write instruction has to be executed (reading from the bus and writing into the buffer) per byte. This results in a computation time of c = 1 MByte/75 MHz*2 = 14*2 = 28 ms. It is also assumed that the maximum event rate for the 'Find_route' use case is the inverse of the deadline, i.e. the period is p = 15 s. Furthermore, it is assumed that the amount of data that is necessary to transfer the route is negligible in comparison to the map. This results in a processor utilization of $c/p = 0{,}028/15 = 1{,}9.10^{-3}$.

e) For reasons of simplicity, the 'Speaker' task, that is only executed sporadically, is neglected.

To verify the schedulability, Rate Monotonic Analysis (RMA), as e.g. described in [16], is used. The schedulability criterion for a set of n preemptable tasks with computation time c_i and period p_i is

$$\sum_{i=1}^{n} c_i / p_i \leq n\left(2^{1/n} - 1\right).$$

Scheduling of the above four tasks results in a processor utilization of approximately 0,65 < 0,75. Hence the 'Controller' tasks are schedulable on a single processor.

3. CONCLUSIONS

Although architectures become more and more important at all levels of ICT-systems and for all types of stakeholders, we are just beginning to understand the requirements for designing a good architecture. Up to now, architectures have concentrated mainly on the structure of a system and the interfaces. For complex systems, it is, however, also important to deal with the behavior of the system and the interaction sequences between the various components. In general, the types of (orthogonal) design dimensions needed, the views that different stakeholders have onto an architecture and the relation between these two aspects need to be investigated in much more detail before one can answer important questions like:

a) How can we tell whether a project requires a major effort in architecting?
b) How are architectures developed, e.g. who are the relevant stakeholders and decision makers, what strategies are to be followed (top-down, bottom-up or a mix thereof) and what should a design methodology for architectures look like?

c) How should we deal with the product-process interdependence during the development of an architecture?
d) Along which, preferably orthogonal, design dimensions should an architectural description proceed?
e) How can an architecture be described, e.g. in natural language, in graphical form, by formal methods, or a mix thereof?
f) What views on an architecture are necessary to support the different stakeholders in a consistent way?
g) How can we assess the completeness and consistency of an architecture?
   - What criteria should be used to validate an architecture (against the expectations of its users) and to verify it (against the requirements)?
   - To what extent can formal methods support the architecting process and how to verify an architecture otherwise?
h) How can we assess the quality of an architecture? What are suitable metrics and methods for the assessment of the various characteristics of an architecture like reusability, scalability, openness (compliance to standards), etc.
i) How can we manage the architecting function in an organization?

To make the situation even more complicated, each of the above questions has to be solved at different levels of the system hierarchy:

a) Individual systems,
b) system families that seek to minimize the effort associated with variety as described in [11] and
c) standard platforms and architectures (e.g. OLE, CORBA, ODP, TINA, etc.) that try to minimize the development effort by providing standardized abstractions together with a well defined set of services.

Each level has different demands on an architecture and will therefore possibly result in different development activities and evaluation criteria.

The first part of this chapter concentrated on the paradigms for developing a good architecture and especially on the specification of the behavior and the non-functional properties like dependability and other X-abilities. In addition, an attempt was made to identify the relevant design dimensions, the views of the most important stakeholders of an architecture and the most urgent open research questions. The challenge will be to integrate these often-contradicting aspects into a consistent framework, to formalize this framework and to devise the corresponding methodologies.

In the second part of this chapter, an approach for achieving composability in resource constraint component architectures was proposed. This method comprises an extension of the component interfaces that allows

also the specification of resource requirements and interaction styles. The Car Navigation System example demonstrated this approach with respect to the verification of timing requirements. Due to the moderate size of the example, checking that the architectural styles of the components match is straightforward. Although the approach described is not formal, it paves the way for mathematical specification and verification of the composability of software components.

This chapter concentrates on Embedded Systems. Nevertheless, many arguments also hold for the development of administrative systems. An important difference, however, is that for Embedded Systems, usually the worst-case behavior is most critical whereas for administrative systems, usually the average behavior is most interesting.

Although composability is a key feature for the construction of predictable and robust systems [15] and for improving the software productivity by means of component-based software engineering, it is still not paid sufficient attention in practice. This is astonishing since the basic principles have been formulated long time ago (see e.g. [23] and [24]). It remains thus a research challenge to develop suitable methods and techniques that support the construction of compositional architectures. This challenge becomes even greater if formal specification and verification techniques are considered. The proper specification and verification of components and dependability constraints needs further investigations.

Also the specification and implementation of soft real-time constraints must be investigated as extensively as this has been done for hard real-time constraints. This requires a concept that integrates functional and non-functional properties of a system into one semantic framework. Based on such a framework, also appropriate specification, design and verification tools can be implemented. An example of such a formal framework for the specification and verification of the functionality and the end-to-end timing constraints of real-time systems can e.g. be found in [29].

ACKNOWLEDGEMENT

I am grateful for the help of Michel Chaudron in working out the example described in subsection 2.4.

4. REFERENCES:

1. Len Bass, Paul Clements, and Rick Kazman, Software Architecture in Practice, Addison-Wesley, 1998.
2. L. Bergmans and M. Akşit, Composing Synchronization and Real-Time Constraints, to be published in Journal of Parallel and Distributed Computing, September 1996.
3. Barry Boehm and Chris Abts, COTS Integration: Plug and Pray?, IEEE Computer, January 1999.
4. Grady Booch, James Rumbaugh, Ivar Jacobson, The Unified Modeling Language User Guide, Addison Wesley, 1999.
5. R.J.A. Buhr and R.S. Casselman, Use Case Maps for Object-Oriented Systems, Prentice-Hall, 1996.
6. Bruce Powell Douglass, Doing Hard-Time: Designing and Implementing Embedded Systems with UML, Addison-Wesley, 1999.
7. K.H. Ecker and D.K. Hammer, A Polynomial Time Scheduling Algorithm based on Refinement of Schedules, Int. Conference of the Institute for Management Sciences (TIMS XXXII), Anchorage, Alaska, USA, June 1994.
8. F. Ehrens, The Synthesis of Variety, PhD Thesis, Eindhoven University of Technology, 1996.
9. D. Garlan, R. Allen and J. Ockerbloom, Architectural Mismatch: Why Reuse is So Hard, IEEE Software, November 1995.
10. Dieter K. Hammer, Andrew A. Hanish and Tharam S. Dillon " Modeling Behavior and Dependability of Object-Oriented Real-Time Systems ", Special Issue on Real Time Object-Oriented Systems of the Int. Journal of Computer Systems Science and Engineering (IJCSSE), Jan. 1998.
11. H. Hegge, Intelligent Product Family Descriptions for Business Applications, PhD Thesis (in dutch), Eindhoven University of Technology, 1994.
12. Recommendation Z.120, Message Sequence Charts, International Telecommunication Union, Geneva 1996.
13. Recommendation Z.120, Message Sequence Charts, Proposal, International Telecommunication Union, Geneva 2000.
14. I. Jacobson et al., Object-Oriented Software Engineering: A Case driven Approach, Addison-Wesley, 1992.
15. Krishna Kavi, James C. Brouwne and Anand Tripathi, Computer Systems Research: The Pressure Is On, IEEE Computer, January 1999.
16. M. Klein e.a., A Practitioner's Handbook for Real-Time Analysis, Guide to Rate Monotonic Analysis for Real-Time Systems, Kluwer, 1993.
17. Herman Kopetz, Real-Time Systems: Design Principles for Distributed Embedded Applications, Kluwer, 1997.
18. Herman Kopetz, The Time-Triggered Architecture, Proc. First IEEE Int. Symposium on Object-Oriented Real-Time Distributed Computing (ISORC), Kyoto, April 1998.
19. Philip Kruchten, The 4+1 View Model of Architecture, IEEE Software, Vol. 12, No. 6, November 1995.
20. Philip Kruchten, The Rational Unified Process: An Introduction, Addison-Wesley, 1998.
21. J.C. Laprie, Dependability: Basic Concepts and Terminology, Springer, 1992.
22. Bertrand Meyer, Object-Oriented Software Construction, Prentice-Hall, 1997.
23. David. L. Parnas " On the Criteria To Be Used in Decomposing Systems into Modules, Comm. ACM, vol. 15, no. 12, pp. 1053-1058, December 1972.

24. David. L. Parnas, On the Design and Development of Program Families, IEEE Transactions on Software Engineering, vol. SE-2, no. 1, pp. 1-9, March 1976.
25. Dewayne E. Perry and Alexander L. Wolf, Foundations for the study of Software Architecture, ACM Sigsoft Notes, Vol. 17, No. 4, October 1992.
26. Eberhardt Rechtin and Mark Maier, The Art of Systems Architecting, CRC Press (London), 1997.
27. S. Ren, G.A. Agha and M. Saito, A Modular Approach for Programming Distributed Real-Time Systems, to be published in Journal of Parallel and Distributed Computing, September 1996.
28. Herbert A. Simon, The Science of the Artificial, MIT Press, 1996.
29. Alexei Sintotski, Dieter Hammer, Jozef Hooman, Onno van Roosmalen, DEAL: an Object-Oriented Real-Time Language, to be submitted.
30. Mary Shaw, Larger Scale Systems require Higher Level Abstractions, ACM Sigsoft Notes, Vol. 14, No.3, May 1989.
31. Clemens Szyperski, Component Software: Beyond Object-Oriented Software, Addison-Wesley, 1998.
32. J.P. Verhoosel, D.K. Hammer, E.J. Luit, L.R. Welch and A.D. Stoyenko, A Model for Scheduling of Object-Based Distributed Real-Time Systems, Journal of Real-Time Systems, Vol. 8, Nr. 1, Jan. 1995.

Chapter 4

COMPONENT ORIENTED PLATFORM ARCHITECTING FOR SOFTWARE INTENSIVE PRODUCT FAMILIES

Initial experiences with component frameworks and platforms from the consumer appliances and medical equipment domain

Henk Obbink, Rob van Ommering, Jan Gerben Wijnstra and Pierre America
Philips Research Laboratories Eindhoven, Prof. Holstlaan 4, 5656 AA Eindhoven, Henk.Obbink@philips.com

Keywords: Component, platform, architecture, architecting, embedded software, product families, domain engineering, product lines, product families, component frameworks, platforms

Abstract: Platform-based product families are strategic business assets. A product platform represents a corporate asset from which streams of derivative products of a large variety can be derived and developed (so-called product families). Platform based development promises to be very effective in decreasing development cost and lead times while at the same time increasing product quality and market diversity. Currently industry is in the process of adapting this approach. In the course of time, electronic products have become software intensive. Unfortunately, software engineering processes and technologies that have been developed until now were mainly concerned with the creation of one product at a time. They do not address well the need for development and maintenance of a product platform and its derivative products. In this chapter, it will be shown how component oriented product family architectures provide a promising development paradigm. This paradigm solves the inherent dilemma of the need for careful engineering versus rapid realisation of a large variety of product instances. The approach is illustrated using examples from the medical and the consumer domain.

1. INTRODUCTION

Industrial organisations are continuously striving to improve their product creation capabilities in order to deliver products (goods and services) on global markets, meeting customer needs, exceeding customer expectations, with minimal defects, for the lowest life cycle costs, and in the shortest time.

On many markets, the product's economic life is becoming shorter and shorter. In order to survive in these markets it is amongst others required to combine and balance the need for a careful engineering approach (to guarantee high quality products) with the need for rapid product delivery (to guarantee short time-to-market) [37]. In this chapter we will focus on the software part of the overall product creation process (PCP) and describe a software development approach to achieve high integral quality and timely products on a large variety of markets.

We use the classification scheme of qualities from [3]. These high integral-quality and timely products should be balanced by the following four product quality aspects:

- Qualities that affect the *business performance*: e.g. time to market
- Qualities that characterise the *development process performance*: e.g. reusability
- Qualities that affect the user perceived *product performance*: so-called product qualities: e.g. safety
- Qualities that are *intrinsic to the product*: e.g. conceptual integrity of its software architecture

In [27] five types of product creation processes for product families were identified. See Table 1.

Table 1: Life-cycle characterisation of product creation processes

PCP	Characterisation of the Product Creation Process	Product Life-Cycle Phase
0, 1	The PCP enables the creation **OF** *new* families	Embryonic
2	The PCP enables the creation **FOR** *new* families in a known domain	Growth
3	The PCP enables the variation **ON** *existing* families	Mature
4	The PCP enables the variation **WITHIN** *existing* families	Ageing

These 5 types can be mapped [2] on the standard industry/product lifecycle, shown in Figure 1:

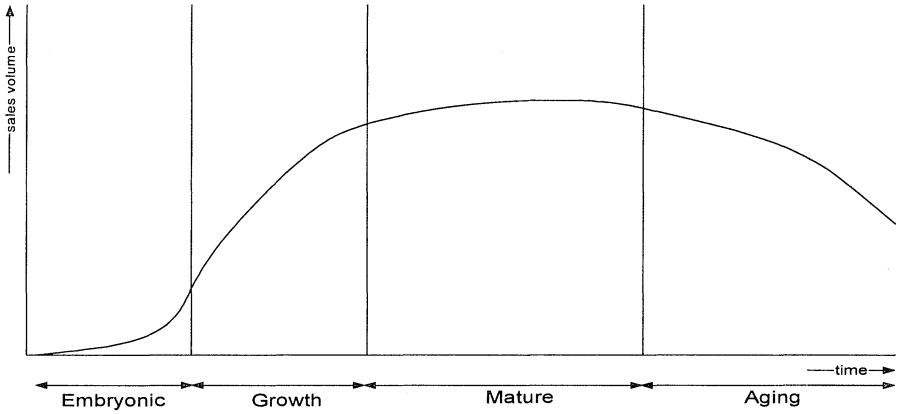


Figure 1: Product lifecycle

In [28] the results were shown of the analysis of the software architecture on a number of Philips product families covering all the above mentioned phases, both in the consumer and professional domain. Subsequent assessment of these product families has led to the conclusion to renovate the software of two product lines in the mature phase of Figure 1. In this chapter, we will describe our initial experiences with this renovation, using a component oriented approach. One product family is from the medical equipment domain (capital goods) the other product family is from the consumer appliances domain (consumer goods).

In Section 2, we describe the major principles underlying our product family engineering approach. Section 3 presents an overview of the approach. In Section 4 we describe our approach to domain engineering. Section 5 explains our family architecting approach. Then in Section 6 we describe experiences in the medical domain while Section 7 reports our experiences in the consumer domain. In Section 8 we present a discussion and some conclusions and finally in Section 9 some further research is indicated.

2. PRODUCT FAMILY ENGINEERING PRINCIPLES

2.1 Introduction

In [8] 201 software-engineering principles are described. In this section, we describe nine product family engineering principles that have been proven useful in the conception and development of the approach used:

1. Domain orientation
2. Integral quality
3. Dependent independence
4. Architecture centric
5. Uncoupling domain solution capabilities
6. Hardware abstraction
7. Targeted value at low cost
8. Design for change
9. Co-operating in separation

In the subsequent sections, we will shortly describe each of these principles.

2.2 Domain Orientation

The product family concept is by its very nature domain specific. We use the definition as originally given by the Software Engineering Institute.

> A software product-line/-family is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission [6].

The experiences described are obtained from projects in the medical and the consumer application domains. In both domains, a large variety of products and corresponding product families exist. In the consumer domain for example we find products like VCRs, TVs, telephones. In the medical domain, we find products like ultrasound equipment, computed tomography equipment, magnetic resonance equipment, and x-ray equipment.

One of the fundamental issues that have to be tackled is the scoping of the domain. One of the heuristics that has proven useful is that of "Think big act small".

2.3 Integral Quality

As already stated a major challenge for industry is a timely delivery of a continuous stream of high integral quality products for a multitude of customers on global markets.

Leading industrial practice is to use platform and product family strategies to achieve these goals. In that way, it is possible to maximise the commercial diversity and minimise the technical diversity of a product family at a reasonable cost. At the same time, in order to achieve integral optimal product and process quality, a development approach is required that optimises, integrates and balances several qualities simultaneously, to satisfy the numerous stakeholders [10]. More specifically in order:

- To optimise and balance the *development process performance* qualities like development cost, modifiability, portability, reusability, integrability and testability, we decided to use a component paradigm [36] within an overall reuse [17] and platform [25] strategy for developing the software. See Section 2.4 and Section 2.8.
- To optimise and balance the *intrinsic product* qualities like conceptual integrity, correctness, completeness and buildability we adopted a strong architecture approach [33], [21], [35], [13]. See Section 4.
- To optimise and balance the user perceivable *product performance* qualities like performance, security, availability, functionality, usability, error handling, satisfaction, again a strong architecture focus was needed and is described in Section 4.
- To optimize and balance the *business performance* qualities like time to market, product cost, projected lifetime, targeted market, rollout schedule, the above mentioned ingredients are integrated using a platform [25] based reuse strategy [17] for product families [29], [12], [37]. See Section 2.10.

For each product family a particular "optimal" mix and match of the above mentioned qualities has to be determined. We call this the required integral quality profile or quality footprint of the family. This profile determines which tradeoffs, choices, and decisions have to be made during the architecting process.

We will see that despite the superficial similarity the difference in these requirements lead to interesting variations of the product family engineering approach. This will be explained in Section 6 and Section 7 respectively.

2.4 Dependent Independence

Components are the basic building blocks of our products, product families, platforms and architectures.

> A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties [36].

More precisely a component is the *independent* unit of deployment. It is the smallest piece of software that can be combined into different product configurations. In the software architecture, components are the units of software reuse, the units of configuration management, the units of testing, etc. A component implements one or more interface specifications. A component has an *independent* lifecycle from its context.

An *interface* is the unit of specification. It is independent from any particular implementation or component. One interface is often implemented by multiple different components, of which one, more or all components may be present in any single product. In the software architecture, interfaces are the key enabler of reuse.

For a component to be independently deployable, the component needs to be well separated from its environment and from other components, characterised by properties like self-containment and independent lifecycle. A component therefore encapsulates its constituent features. In addition, since it is a unit of deployment, a component will never be deployed partially. Third parties don't expect to have access to the construction details of all the involved components (except for documentation purposes).

For a component to be composable with other components, the component needs to be sufficiently self-contained or *independent*. In addition, it needs to come with clear specifications of what it *requires* and *provides*. In other words, a component needs to encapsulate its implementation and interact with its environment through well-defined interfaces, or explicit *dependencies*. This leads us to the principle of *Dependent Independence*.

In the Sections 6 and 7 of the chapter it will be explained which different decisions had to be taken in the two domains to arrive at the particular component models used there.

2.5 Architecture Centric

In the previous section is has been argued that components in products interact. The architecture is the notion that enables us to reason about the

meaningful combinations and interactions in a systematic way. Moreover the benefits of a good architecture [21], [35], [3] are manifold. A good architecture allows to ensure product features, control complexity, make explicit decisions, make tradeoffs, manage evolution, organize development, support system families and large scale reuse and address the interest of the various stakeholders in a systematic way. Architecture allows us to look at a *system as a whole* taking the different viewpoints and concerns from the various stakeholders with their potentially conflicting quality requirements from section 2.3 into account.

The field of software architecture research is relatively new, but there is no lack of definitions for software architecture. We use the definitions as proposed in [14]:

- **Architect:** the person, team or organisation responsible for systems architecting.
- **Architecting:** the activities of defining, documenting, maintaining, improving and certifying proper implementation of an architecture.
- **Architecture:** the fundamental organisation of a system embodied in it's components, their relationships to each other and to the environment and the principles guiding its design and evolution.

Architecting is about providing systems concepts, principles and rules. It also addresses system structuring, making decisions and trade-offs. The main challenge for the architect is to blend the "user" needs and the available means (technologies, processes, and skills) into a feasible solution in a way that satisfies the overall business needs.

2.6 Uncoupling Domain Solution Capabilities

The two application domains described in Section 2.2 contain very complex products. These products are the result of the combination of a number of generic solutions that have evolved in the particular domain. These solutions comprise skills, technologies and processes; together we call them *domain solution capabilities*. These domain solution capabilities can be ideally structured using a typical domain specific "hierarchy"[1] as shown in Figure 2. The example is taken from the HP inkjet product family as documented in [25]. In the centre of this picture we see the core capability from the domain: e.g., ink processing and paper processing in the printer

[1] In practice the structuring of the domain solution capabilities need not to be a pure hierarchy, although this would break the desired uncoupling. An interesting example is the blurred interface between embedded software and electronics for HW-SW codesign [34].

domain, audio processing and video processing in the consumer domain, medical image processing and patient handling in the medical domain.

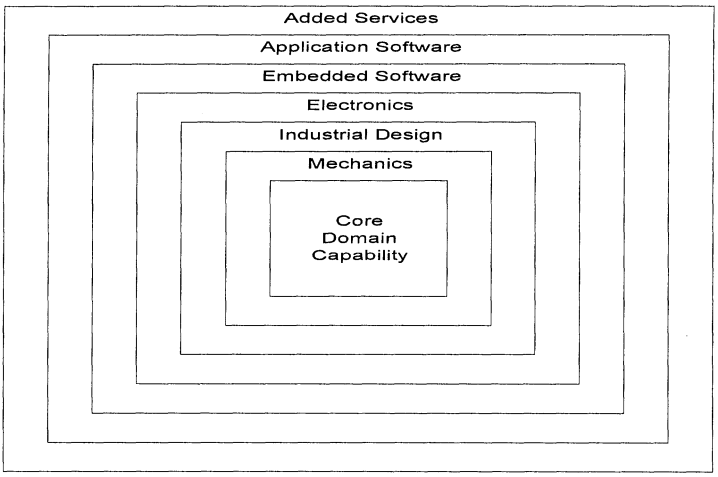


Figure 2: Hierarchy of domain solution capabilities

Around these core domain solution capabilities, we see a set of key capabilities: mechanical design, industrial design, electronic design, embedded software, application design and service design. For the two application domains discussed in this chapter we discuss the renovation of the embedded software, application software and service parts.

Each of the domain solution capabilities in Figure 2 preferably has its own life cycle. Good modular product architectures provide uncoupling of the major domain solution capabilities in order to enable their independent evolution. In Section 2.7, a component oriented reference architecture will be shown that deals with some of these concerns.

The ordering and structuring of the domain solution capabilities into a conceptual framework as described above is the work of an experienced system architect in the particular application domain and is guided by experience, best practices, regulations, conventions, … both in the application domain and in the solution domain. For a discussion of the latter two notions, we refer to Section 4.1.

2.7 Hardware Abstraction

Through the fundamental choice for components in Section 2.4 our architectures are by their nature component based and through the focus on the consumer and medical domain they are by nature application domain-specific. The terms *domain-specific-architecture* or *reference architecture*

[13] will be used to define an architecture that is generally applicable in a particular domain. In this type of architecture the concepts, decisions, structures, rules, interfaces, reach as far the scope of the domain is defined. It is often shown how the domain functionality is mapped to the architectural components. A *product-family architecture* is similar to a domain-specific-architecture, but its influences also reach out to all the members of the family.

In this chapter, we deal with software intensive systems. We therefore want to abstract as much as possible from changes in the other solution domain capabilities, either through very small interfaces or through explicit abstraction interfaces. One of the changes we want to isolate is the quickly changing hardware. We want to abstract from two major hardware concerns, the domain hardware concerns and the computing infrastructure concerns, each having their independent lifecycles. In Figure 3, we present the generic architecture that holds for most software intensive products in both application domains.

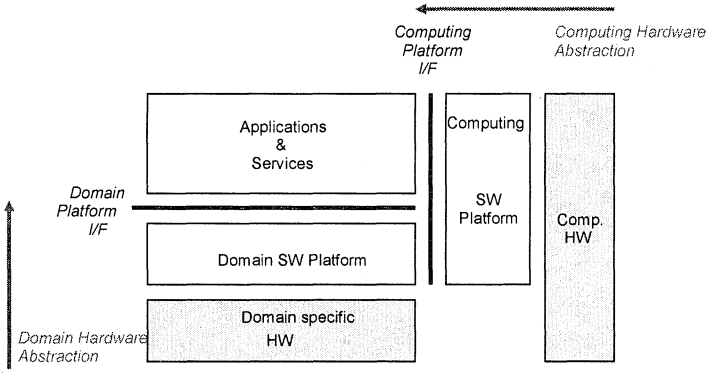


Figure 3: Domain specific reference architecture

In Figure 3, we see abstraction of the two hardware types through two different software "platforms". The platform notion will be explained in the Section 2.8.

2.8 Targeted value at low cost

Companies can have three basic, often incompatible, strategies to achieve the following market leadership goals:

1. They can strive for cost leadership. Provide the cheapest product.
2. They can strive for value leadership. Provide the most valuable product.

3. They can strive for customer intimacy. Provide the product that is optimally tuned towards the need of the customer.

Using classical development approaches it is often only possible to achieve leadership along one of the above mentioned goals.

In order to obtain *value* AND *cost* leadership the platform concept has proven to be applicable [18].

> A product platform is a set of subsystems and interfaces that form a common structure from which a stream of derivative products can be efficiently developed and produced [25].

Product families on the other hand are needed to enable the *targeted* commercial diversity that is required to achieve the third goal.

To achieve leadership along the above mentioned three goals simultaneously, it is necessary to combine the concepts of the previous sections into the following modified platform concept.

Our notion of platforms has an architecture that integrates the notion of components from Section 2.4, with the architecture notions from Section 4. In the products discussed in this chapter various technologies like mechanics, electronics, and software are present and are packaged in self-contained functional components or subsystems. Each of these subsystems has clear interfaces, be they physical, electronic or software-based to other subsystems. The platform architecture as a whole has also interfaces to the external environment. In order to be more flexible we have to relax the requirement of a common structure. We therefore use the following adapted definition.

> A product family platform is a set of (component- based) subsystems and interfaces (with their associated processes, documentation and tools) from which a stream of derivative and composite products (families) can be developed and produced according to a domain specific architecture or product family architecture.

All the planned and targeted family products are described by a so-called product family map shown in the upper part (the time dimension is not shown for simplicity reasons) of the so-called power tower in Figure 4 [25]. In the middle part, the successive platform generations are shown. In the lower part the core skills, competencies, and capabilities of the company that fuel the platform are depicted. They often correspond with the core competencies of a particular company and comprise for instances the solution domain capabilities of Section 5.

Product platforms can have various degrees of integration. In some cases a lot of common functionality, across the different member of the family, has

already been pre-integrated and tested. In other cases, the platform is just a set of components (intellectual property) that can be combined according the platform specific architecture composition rules.

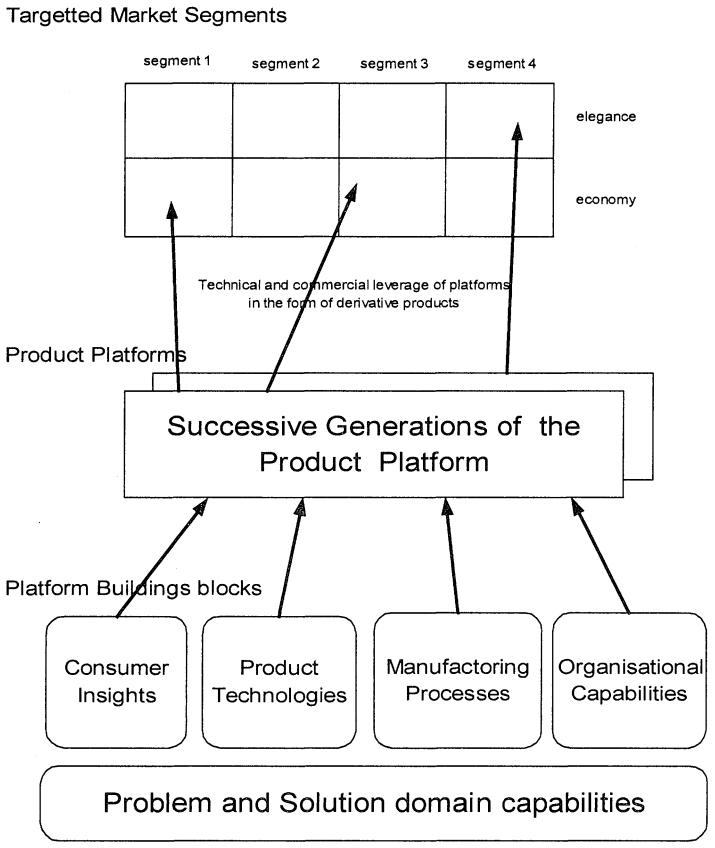


Figure 4: The power tower: an integrative model of product and process innovation

2.9 Design for Change

Because we are dealing with software intensive product families, we will now elaborate the "software platform". The software part of the platform can be seen as the design and implementation of a "domain- machine", a core set of software subsystems that "propel" the entire system. The variability and the flexibility of a software platform is determined by the mechanisms that are provided to combine (compose) the various components and interfaces during development or use of the systems derived from the platform. Components are developed and can be plugged into this domain machine to

provide a variety of products, either as new versions of existing products or as entirely new derivative products.

The integration of subsystems is greatly facilitated if the internal interfaces between platform components are clearly defined and use agreed (standard) mechanisms where they are available.

The interfaces of the software platform are the key enablers, the source of its power. They capture the "areas" in the platform that are "stable" under the tremendous change pressure from several sources.

We can at least distinguish the two major types of interfaces:

Internal platform interfaces between platform components. The mechanisms by which the key subsystems of the platform engine (domain machine) interact with each other.
External platform interfaces. Between the platform and external systems (including users) interacting with them.

In addition, we distinguish:

Extensibility interfaces. Interfaces provided by the platform to enable the required variability and extensibility for add-in components (plug-ins, applications)
Hardware abstraction interfaces. Interfaces that enable the domain machine to run on a variety of domain-specific hardware and general purpose computing hardware.

2.10 Co-operating in Separation

In [17] and [22] it is argued that architecture centric reuse is one of the most promising reuse strategies at this moment.

Architecture centred reuse is best organised as three separate, but tightly co-operating types of processes:

- *Application Family Engineering* (AFE). The AFE-process (one per platform) is responsible for the initial definition and future evolution of the platform's architecture. This of course in the context of a normal product planning and product decomposition process. (These encompassing processes are not shown in Figure 5). The AFE process comprises the decomposition of the architecture into a number of major components and the definition of the interfaces between these components.
- *Components System Engineering* (CSE). The CSE-processes (one for each component) develop and maintain the various platform

components as defined by the AFE-process and sometimes integrating and testing the common core components into a product platform.

- *Application System Engineering* (ASE). The ASE-processes (one for each range or line of products) develop the products, (re) using the components, or platform (developed by the CSE-processes).

These three processes can be easily mapped onto the dual For-Reuse and With-Reuse lifecycles [20], in which domain engineering (domain analysis, domain design, and domain implementation) is uncoupled from application engineering (application requirements, application design, and application implementation). The AFE and CSE processes are part of the domain engineering activities, while the ASE process is the same as application engineering.

In Figure 5, the relevant platform engineering processes are shown, with the key deliverables: Domain Terminology, Reference Requirements, Reference Architecture, and Reusable Components.

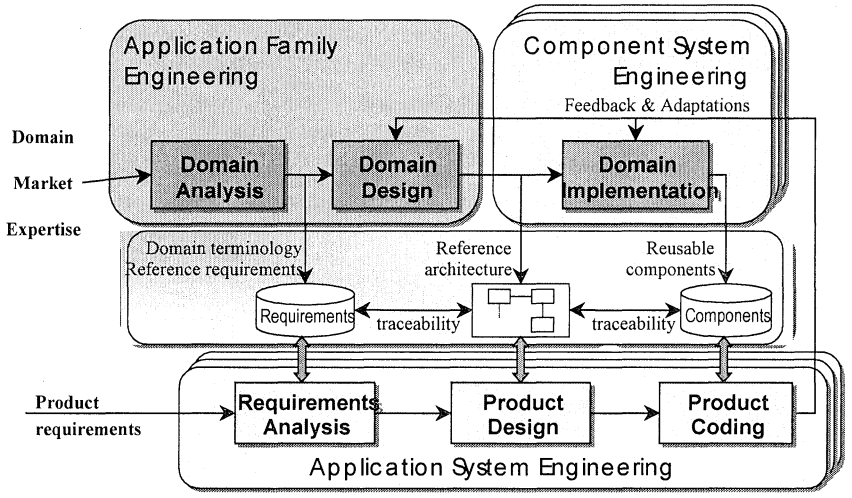


Figure 5: Platform engineering processes

2.11 Product family engineering principles and approach

The above mentioned principles contribute:

- domain modelling
- emphasis on architecture to achieve quality attributes, to reason about the system as a whole, and to provide a framework for components

- use of components to enable independent evolution
- uncoupling of different domains, and abstraction of domain hardware and computing infrastructure
- use of product platforms to address multiple market segments
- use of interfaces to enable controlled extension, variation and change
- a hierarchical platform engineering process

Together, these contributions constitute the key elements of our product family engineering approach, discussed in the subsequent sections.

3. PRODUCT FAMILY ENGINEERING APPROACH

This section sketches the process for our product family engineering approach, incorporating the "Co-operating in Separation" principle. It provides steps that are more detailed. We have divided the overall process in a number of subprocesses, see Figure 6. It is important that separate people or projects are responsible for these subprocesses so that they check each other. The co-ordinating part is the family engineering process (previously called Application Family Engineering or AFE). When family engineering has proceeded far enough, it is possible to start platform and product engineering (previously called Component System Engineering, CSE, and Application System Engineering, ASE).

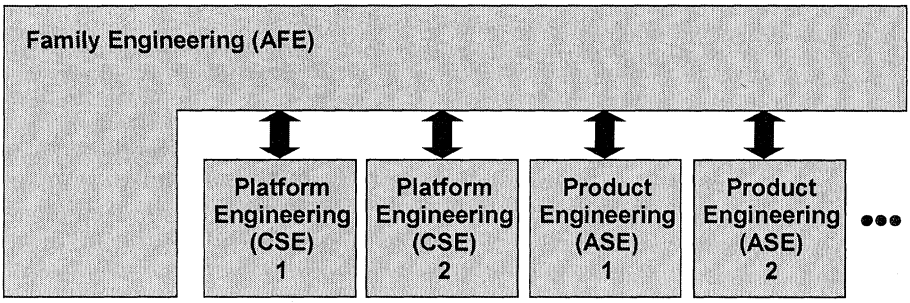


Figure 6: Top Level Processes

Platform engineering develops platform components and product engineering develops products using these platform components. Both of them receive guidance from and provide feedback to the family engineering process. Apart from that, these two kinds of subprocesses can typically be organised along the same lines as in single product development (see, e.g.,

[16]). Note that often there is one platform engineering subprocess (although there could be more), but there are typically several product engineering subprocesses, either in parallel or consecutive.

The family engineering process is different because it explicitly deals with the family development. We propose to subdivide it into activities as shown in the following table.

Table 2: Family engineering activities and results

Activities	**Results**
1. Defining the process	Detailed process definition
2. Informal domain analysis	Business context and scope
3. Requirements specification	Use cases
4. Analysis	Analysis object model
5. Conceptual architecting	Reference architecture
6. Defining the product family	Precise family definition
7. Family architecting	Product family architecture
8. Support and supervision	Integrated and targeted products

These activities are not performed sequentially, but in parallel, although they typically have different intensities at different times during development (see [16]). In the rest of this article, we concentrate on the five activities from requirements specification to family architecting. The informal domain analysis activity is very important for family engineering, but a detailed account is outside the scope of this article, so we refer the reader to [9].

We groups these five activities under two headings: Domain Engineering and Architecting for product families to be discussed in Sections 4 and 5.

4. DOMAIN ENGINEERING

4.1 Domain Engineering Concepts

One of the most valuable principles in developing a product family is *domain engineering*. Here we define a *domain* as a conceptual space of possible systems. In most cases, this space is multidimensional and very large, possibly infinite. The members of our product family or population

form a carefully chosen, typically finite, subset of this space. Now the principle of domain engineering tells us to devote our attention in the development process as much as possible to the domain as a whole, and as little as possible to the individual systems inhabiting it. Below we will describe in more concrete terms what this means for the various development activities.

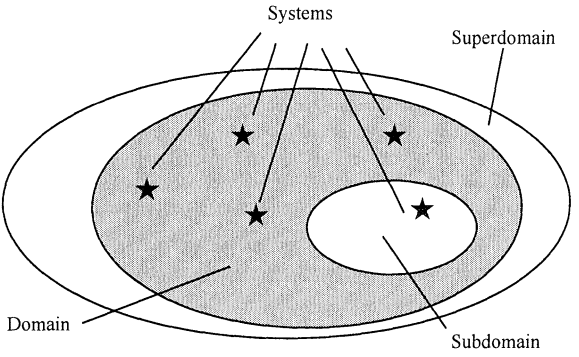


Figure 7: Domains and systems

Several other concepts are important in domain engineering. First, the *scope* of a domain is a characterisation of the extent of the space of possible systems, or in other words, a definition of which systems do and which systems do not belong to the domain. In this definition, the systems can be characterised based on their functionality, but also based on cost, performance, or other non-functional aspects. For example, one may define a domain of TVs that cost less than $500, or a domain of MR (Magnetic Resonance) scanners with a magnetic field of more than 1 Tesla.

In this context, it may be useful to divide a domain into *subdomains*. Here a subdomain can be a subspace of the original domain, in the sense that it contains a subset of the original set of possible systems (we could call this a *system subdomain*), but it can also consist of specific subsystems that are used as building blocks for the original set of systems (a *subsystem domain*). An example of the first would be to divide the domain of medical imaging systems into subdomains such as CT (Computed Tomography), MR, and X-ray systems. An example of the second would be to consider the subdomain of gradient subsystems used in MR (the subsystems that cause a precisely controlled gradient in the magnetic field). Conversely, a domain that contains another domain is called a *superdomain*.

Let us use the term *primary domain* for the domain that matches the scope of the development for a product family at a given time. When starting

a product family development effort, it is extremely important to quickly gain a good idea about the scope of this primary domain, so that the solutions developed are not overly generic or too specific. At the same time, it is useful to explicitly identify relevant subdomains and superdomains, so that potential future extensions and specific optimisations are not blocked by decisions that only apply to the original domain. For example, when developing a family of TVs, one should also consider the population superdomain that also contains VCRs or even TV/VCR combinations, because they might all use the same tuner components. On the other hand, by considering the subdomain of TVs for the North American region, one can devise specialised solutions for solving some image quality problems that plague the NTSC signal format.

Within a domain, the diversity among the systems inhabiting it is localised in certain *variation points*. (More precisely, we define a variation point as a single dimension in the multidimensional domain space.) This diversity can have different sources. On the one hand, there are the observable features, which may be of a functional or a non-functional nature. On the other hand, the same features can often be realised by different technological options, and these are not directly observable by the user. For example, the possible systems in the TV domain can differ in the size of the picture tube, the number of loudspeakers, the presence of teletext, which are all observable. On the other hand, the TVs can also differ in whether the sound processing is done by a dedicated chip or by a DSP, and this difference is not observable from the outside.

Furthermore both the features and the technology are most likely to change over time, and domain engineering should certainly take that into account. Features change on the one hand because the systems are being used in a new way (e.g., TV channel hopping brought the need for a remote control), and on the other hand because in the market the products need to distinguish themselves from the others in the family and certainly from competitors' products. Technology changes are sometimes forced by the environment (e.g., when a supplier no longer supports the currently used version of the operating system) and sometimes they are enabled by advances in the developing company. Developing and maintaining a roadmap is a good way to ensure that all these changes do not come as surprises but are planned for.

Sometimes a distinction is made between *problem domain* and *solution domain*. Here the problem domain refers to those aspects and entities that are relevant to the users of the systems in the family, while the solution domain covers, in addition, the aspects and entities that are relevant in building the systems. Consequently, this distinction makes most sense for systems where

the users are relatively unfamiliar with how the systems are implemented (typical for consumer electronics).

Domain engineering takes place in all phases of the development of a product line, and can therefore be subdivided into activities such as domain analysis, domain architecting, and domain implementation (See Figure 5). Below we briefly describe our approach to domain analysis, while the next chapter will sketch some elements of domain architecting.

4.2 Domain Analysis

Domain analysis, the first step in domain engineering consists of requirements specification and structural analysis. In modern software development methods [11] [16], requirements specification is mostly done in the form of use cases [15] where the interaction of the users and the system and its effect on the system are described. Several notations can be used for this purpose, ranging from English text to the use case diagrams and sequence diagrams of UML [4].

Mostly this requirements specification activity is followed by structural analysis, typically in the form of object-oriented analysis (OOA) [7]. The result is an analysis object model, often again expressed in UML using mostly class diagrams.

The special thing about doing this for a domain, instead of for a single product, is that the requirements specification and the analysis model cover the whole domain. Furthermore, it is important to precisely express the variation points in the requirements specification as well as in the analysis model. Some techniques for doing this are provided by the FAST method [37]. Especially in our work on professional electronic systems, we have good experiences with a domain analysis approach where we integrate requirements specification with structural analysis [1].

The result of this is a precise and extensive description of the required functionality for all the systems in the domain, which makes it possible to specify an individual system in a very concise way by just fixing the variation points. In this way it becomes very easy to define our product family (as a finite subset of the domain) and to be flexible in changing this definition as circumstances require. Another important point is that, although the approach enables a large degree of diversity between the systems in the product family, it discourages unnecessary diversity.

Finally, the analysis object model forms an excellent starting point for developing a design object model for our product family. Broadly speaking there are two possible approaches for doing this: transformational or elaborative. In the transformational approach, a design model is developed disjoint from the analysis model, but of course based on the information

from the analysis. In the elaborative approach, the design model is developed as an extension of the analysis model. Both approaches have their own advantages, but also their own difficulties [19]. In any case, the detailed way of how the design model is developed largely depends on the architecture, which is the subject of the next section.

5. ARCHITECTING FOR PRODUCT FAMILIES

As explained in the previous section, developing a product family is best done in a domain engineering approach, where the attention is directed at the domain as a whole, rather than at individual systems. This also holds for architecting, the activity of developing an architecture, and therefore what we describe in this section is essentially domain architecting, even though we will often speak of a product family architecture.

5.1 Reference Architecture and Family Architecture

Architecting is making decisions. First, these decisions apply to the structure of the systems to be built. This involves choosing an architectural style (e.g., layered, multi-tiered, blackboard, etc.) and, depending on that style, choosing and identifying the structural elements, such as layers, subsystems, packages, components, and interfaces. (Note that here "identifying" does not mean "defining in full detail". This can be done later by the responsible designer under supervision of the architect.) Second, architecting involves identifying concepts and mechanisms to be used in connecting the structural elements together. Often these concepts and mechanisms take the form of design patterns, but ideally they are also supported by reusable implementations provided by one or more of the structural elements.

Keep in mind that most architectural decisions have a more profound influence on the quality attributes of the resulting systems than on their functionality [3].

In domain architecting, the above decisions can have different scopes. Global decisions apply to all systems in the primary domain and to all structural elements from which they are built. Local decisions apply only to a single system or a single element. Most of the decisions, however, have an intermediate, "regional" scope: they apply to a subset of the systems or to a subset of the structural elements, in other words, to a subdomain. Such subdomains can be defined in various ways, for example:

- **Geographical:** Where are the products sold? This often has implications for the standards and conventions that the products must comply with.
- **Organisational:** Which development department is responsible? Different departments often have their own identities and cultures, and often it is best to respect these rather than to force uniformity.
- **Technical:** What kind of system or element is it? Certain quality attributes can have different importance for different systems or elements, and therefore the architectural decisions determining them may differ.

Obviously, it is important to make explicit the subdomain to which a certain decision applies. Nevertheless, in practice this is often left unclear. Also note that most often architectural decisions should not be mandatory, but should have the character of a guideline, which can be overridden inside a narrower subdomain if there is a good reason to do so.

Together, all these architectural decisions for a domain establish a *reference architecture*. This reference architecture will typically need several views [21] to be expressed completely. In fact, it will sometimes need several views of the same kind to express the different decisions for different subdomains.

When the precise subset of the domain that is to be developed as a product family has been chosen, it is often possible and necessary to make a number of more specific decisions. Since these decisions do not apply to all the systems in the domain, but only to the specific family, we will call them the *family architecture*. Starting from this family architecture, a lot of details must then be added in order to arrive at a design object model, from which the code can be derived in a more or less straightforward way. As mentioned above, the analysis object model must be taken into account here.

Often it is useful to distinguish a *conceptual* architecture and a *technical* architecture. Here by conceptual architecture we mean a representation of the architecture in terms that are independent of the underlying computing infrastructure (operating system, programming language, middleware, etc.), whereas the technical architecture includes the choice of that computing infrastructure and the mapping of the conceptual architecture onto that infrastructure. This same distinction can be made between conceptual and technical design. The advantage of this distinction is that in the conceptual architecture and design one can concentrate on the essentials without being distracted by the many details involved in mapping them onto an operating system, programming language, etc. We can summarise the relationships between requirements, conceptual and technical architecture and design in the following picture.

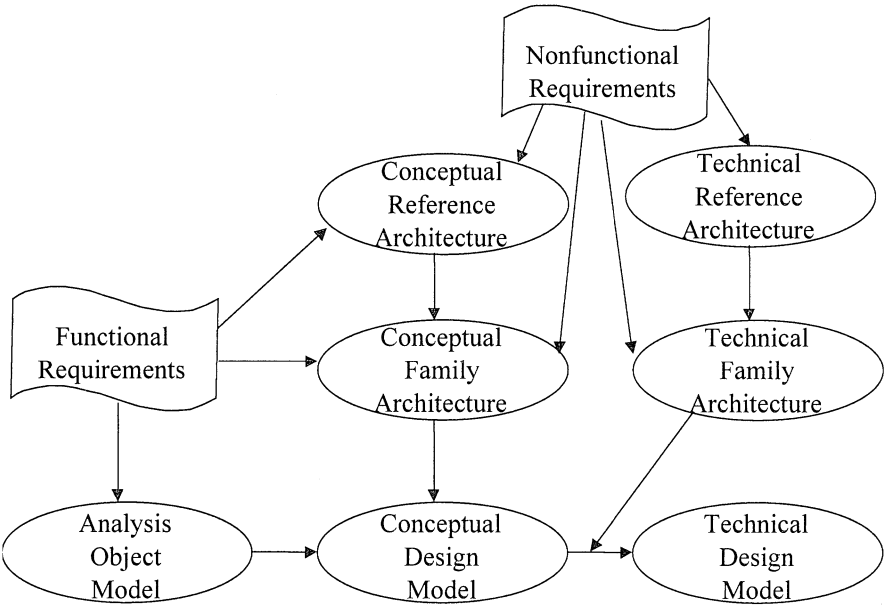


Figure 8: Relationships between requirements, architecture, and design

5.2 Supporting Diversity

One of the most important things to be described in a reference architecture is the roles of the various components from which the systems in the family will be built, and the interfaces between them. Because such a component is a special kind of module (i.e., a package of work for a small group of designers) the usual heuristics apply here, e.g., maximise cohesion and minimise coupling. But here, in addition, the architect must take the diversity in the domain into account. Since we want to build a system in our family by combining a number of components, it is important that each variation point is allocated to a single component, as much as possible. This is because whenever several components are responsible for dealing with a single variation point together, they cannot be deployed independently, or in other words, their meaning as independent units of deployment is compromised. In domains where components can be recursive, in the sense that they can themselves be built up from components, the reference architecture does not have to specify the complete component hierarchy, but may stop at a suitable level.

There are roughly two ways in which a component can handle the diversity caused by one or more variation points:

- The component may have a number of parameters, called diversity parameters, that can be given values according to the choice in these variation points.
- The role of the component itself or of a subcomponent can be played by one or more out of a set of alternative components with (approximately) the same interfaces in both directions. Moreover, components can be connected together in different ways. We call this mechanism a *component framework.*

It is not always very clear which mechanism to choose. A heuristic that often works is to use the diversity parameter mechanism for quantitative diversity and the component framework mechanism for qualitative diversity.

5.3 The Platform Approach

When this subdivision into components has been defined, components can be clustered according to their development life cycles and their usage in the various systems in the domain. If one finds a sufficiently large cluster of components that have similar life cycles and are used in multiple systems in the planned family, this cluster may be developed as a platform. Here not only the architecture counts, but it is also important that the development organisation matches or can be made to match the platform model. Keep in mind that it is possible to distinguish more than one platform wherever that is useful from an architectural and organisational perspective.

In the end, the idea of the platform approach is that individual products can be developed relatively quickly by assembling and configuring a number of components from the platform(s) and possibly adding a few product-specific components.

The platform group develops a platform, which is delivered to the product groups. The platform consists not only of software components, but also of other parts, including requirement and design documentation, interface specifications, architectural rules and guidelines, tools, test environments, user manuals, etc. The platform contains those entities that are relevant for several family members (not necessarily all). The product groups should not modify these entities, but combine and extend them in order to come to a specific family member.

For each family member, the family architecture is the baseline. A platform contains some of the components identified in this family architecture. Some of these components are fixed, i.e., they are generic for all family members. The component frameworks offer the possibility to add specific components to the family member to realise specific behaviour.

Furthermore, diversity parameters are given a value for each specific family member.

In Figure 9 below, the usage of a platform is illustrated. The platform is used to create different products. Of course, when more than one platform were used, the picture would become more complicated. What the figure also illustrates, is that the platform will evolve over time. For example, it may be the case that some software component that was developed for a specific family member becomes applicable for other family members. Then, this software component can be integrated into the platform.

Over time, several releases are made of the platform. Each release contains a set of compatible components. When a component is modified in a new release, a new interface has to be added to the component, since it is not allowed to modify existing interfaces. The component itself keeps the same identity. This way, both existing and new family members can use the same components. An important question is how long old interfaces still have to be supported.

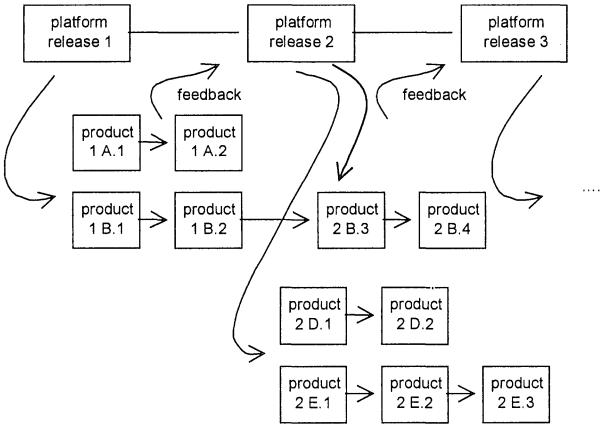


Figure 9: Using the platform

5.4 Architecture Verification

A good software architecture is the basis of each large software system with a long lifetime. A specification of the software architecture of a large system can usually be found in the system design documentation. However, the architecture of the implementation (the implicit architecture) may deviate from its specification in the system documentation (the specified architecture). By means of software architecture verification, it is possible to formally verify in an automated way whether the implicit architecture of a

system is consistent with its specified architecture. This helps to maintain the conceptual integrity of the system [3]. More on architecture verification can be found in [31].

6. EXPERIENCES IN THE MEDICAL IMAGING DOMAIN

6.1 Product Characteristics

Philips Medical Systems is one of the worlds leading suppliers of medical imaging equipment. Its product range includes conventional x-ray, computed tomography, magnetic resonance imaging, and ultrasound equipment. In this section, the experiences of a product family in this domain are discussed. Some of the main characteristics for this product family are:

- Compared to consumer products like televisions and DVD players, as discussed in section 7, only a relatively small number of products are delivered in the field. Instead of the high volumes for the consumer market, only a few thousand products are made. Almost each of these products is different due to high configurability and customisability.
- The delivered products must be maintained for a long time, about 10 to 15 years. Furthermore, updates of mechanical, hardware and software components can be made in the field during the lifetime of a product by field-service engineers.
- New features must have a short time-to-market. It must be noted that products in this domain with digital image processing have been around for more than ten years. Consequently, the basic functionality is not the main issue anymore but the specific features that the various customers request. These additional features add to the complexity of the product family.
- If a product does not operate according to the specification, it may be potentially dangerous to the health of the patients and the personnel. Consequently, high demands are placed on the safety and reliability of the products. For example, the products must be approved by the FDA (Food and Drug Administration) before they are admitted to the U.S. market.
- The product family covers a number of system subdomains. A number of groups exist, each having expertise on such a subdomain. The

development of the various family members is distributed over these groups.

In order to deal with these characteristics, reuse over the family members is needed. This is achieved by developing a common platform for all family members.

6.2 Domain Analysis

Domain analysis has been performed for the medical imaging product family as described in section 4.2 and in [1]. The resulting analysis object model for this family became quite large, containing about 100 class diagrams, 700 classes, 1000 relationships and 1500 attributes.

The general feeling with the project crew is that this way of requirements modelling and specification lays a solid and stable basis of shared knowledge for further development and that the effort was well spent. Even when the specifications changed somewhat during the project, only minor modifications to the object model were necessary.

6.3 Product Family Architecture

Based on the products that will be part of the family, a product family architecture is defined. For this architecture, a number of architectural principles have been identified [32]. Summarising, the main principles are:

- Layered Architecture

  The system is decomposed in a number of layers. The main layers are Infrastructure, Technical and Application. The Application layer contains the application knowledge, e.g. the procedures to acquire images, and analytical functions. The Technical layer provides an abstraction of the underlying hardware to the Application layer, e.g. image processing functions. The Infrastructure layer provides basic facilities to the other two layers, like logging and field-service facilities. These three layers are internally decomposed further.
- Independence of Units

  The system is decomposed into a number of units. A unit contains a coherent set of functionality, and deals with a subsystem domain of the complete identified family domain, e.g. acquiring images or processing images. In order to avoid a monolithic design, units should be self-contained and de-coupled.

  With a self-containing unit is meant here that when such a unit is added to the system, no adaptations of other parts of the system are required.

This means for example, that each unit must do its own error handling and logging and provides its own field-service functionality for usage by the field-service engineer. These kinds of functionality are called aspects [26]. Architectural rules concerning these aspects are formulated that apply to the units. Support is provided for realising aspects by the infrastructure layer.
Furthermore, to avoid a monolithic structure, the units should have no direct knowledge of each other when this is not required. To achieve this, a number of concepts are applied, e.g. event notification and facilities based on the blackboard pattern [5]. Various units are connected to such a blackboard facility, which enables these units to achieve combined behaviour without direct interaction between them.

Applying these principles results in a system decomposition into units, divided over three layers, schematically shown in Figure 10. The decomposition of the technical layer is based on the various hardware devices, e.g. image processing hardware, the application layer is decomposed based on the functional areas in application workflow, e.g. acquiring images, analysing images, and the decomposition of infrastructure layer is based on the infrastructure facilities, e.g. logging and field service. The actual product family architecture contains a little over 30 units.

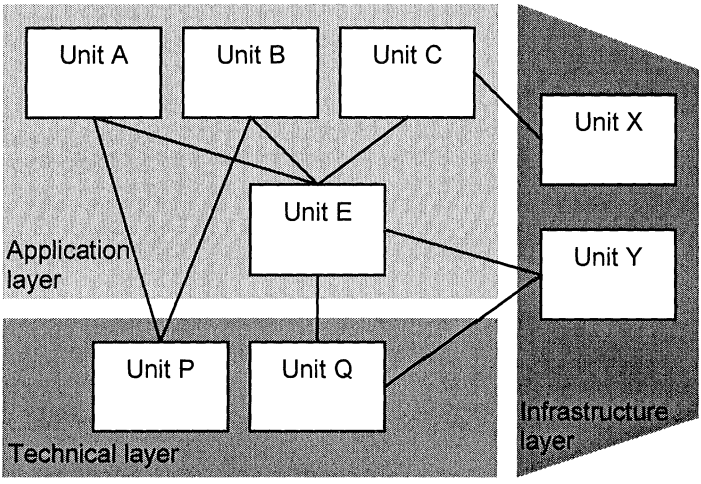


Figure 10: Schematic product family architecture

The analysis object model focuses on the functional requirements; the product family architecture takes all quality attributes into account. Together they form a basis for the design, as described at the end of section 4.2. This

is achieved by assigning classes from the analysis object model to units in the architecture, and the design activity is responsible for defining additional design classes to form a complete design.

6.4 Supporting Diversity

Diversity in the product family is caused by diversity in features and diversity in realisation technology. In our medical product family, examples of diversity in features are different procedures for acquiring images, different ways of analysing images, etc. Examples of technology related diversity are new implementations of image processing hardware and the usage of faster and larger hard disks for image storage. It is important to analyse both the diversity and the commonality between the family members.

The product family architecture presented in section 6.3 only specifies a high level view and applies to all members of the product family, i.e. each unit is in principle present in each family member, although some units are optional. Each unit consists of one or more components. In the previous section, the variance is not made explicit yet. However, each unit may contain variation points at which variation occurs for the various family members.

The issue how to support this diversity is closely related to the following two architectural principles that are defined for the medical imaging product family:

1. binary reuse of components
2. division of the product family development in a generic part and member specific parts

Based on these principles, components frameworks are used as a mechanism for supporting diversity on an architectural level, see also [38]. Next to that, diversity parameters are used for component internal diversity. The component frameworks are for example used for analytical functions; the diversity parameters are for example used to realise country specific setting.

As a result, a generic skeleton [23] is constructed which is based on the unit structure and the generic components within these units. It is possible to construct such a skeleton because the various subsystem domains are relevant for all family members. The generic skeleton is schematically shown in Figure 11. Here, the unit relations and the dark grey parts form the generic skeleton. The light grey parts represent the specific plug-ins that can

be added to the predefined variation points in order to create a specific family member (configuration parameters are not shown graphically).

Standard technology is applied as much as possible, because of benefits like better tool enviroments and available knowledge of the software developers. For this family DCOM has been used as realisation technology for the components, and the interfaces between components are described in IDL.

For a unit that has variation points, either supported by component frameworks or diversity parameters, the functionality it offers depends on the specific configuration of that unit in a family member. Each unit provides an availability interface that allows the clients of that unit to determine the available functionality of that unit configuration at initialisation-time. This way, the clients can determine which functionality they can offer themselves.

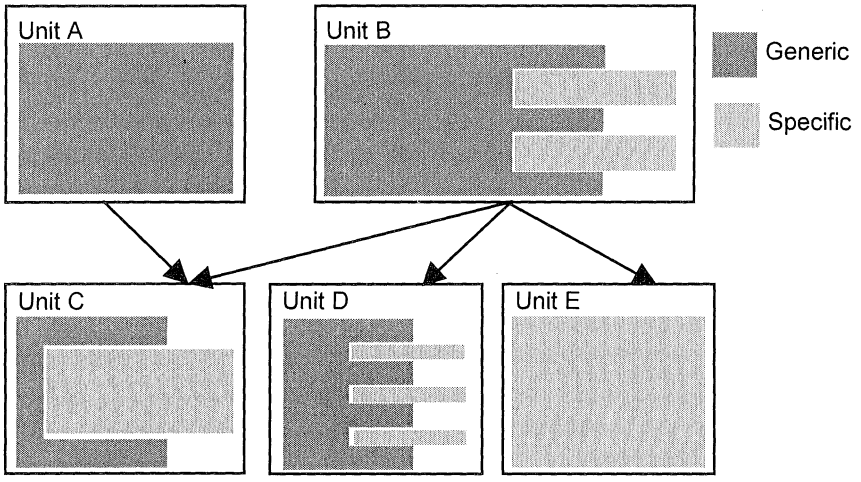


Figure 11: Generic skeleton and specific components

6.5 Platform and Evolution

The platform approach applied for the medical imaging family is as described in section 5.3. One platform group is responsible for the generic skeleton and the specific parts that are relevant for several product groups. The product groups are organised according to market segments, the system subdomains. Such a product group has to configure the system based on the

platform provided by the platform group, augmented with additional components that are specific for a family member.

The evolution of the platform is managed as illustrated in Figure 9. A release of the complete platform is made at a planned moment in time. The advantage of this approach is that the elements belonging to the platform are integrated and tested together in a central place, thus managing the complexity of these tasks and sharing testing effort. Especially for a medical imaging family, where safety and reliability are very important, and where integration testing takes a considerable amount of effort, testing requires special attention. Although the FDA deals with admission of products, not families, it is useful to test the platform as an integrated package, since the platform as a whole can be considered as one of the components from which a specific product is built. This integrated approach is possible since the generic skeleton is shared between the family members, providing a common structure and well defined variation points. An alternative approach would be to release the various units independently, leading to more flexibility. The disadvantage is that less can be said about the integrability of the individual units and test effort must be repeated.

The product family will evolve over time. In the architecture a number of principles have been applied that support evolution of the family. Examples of these principles are layering (to de-couple the application domain from technical domain), independent units (to reduce dependencies) and component frameworks (which allow extension with new functionality).

Interfaces are important entities when considering evolution; when a component is modified, a new interface has to be added to the component. The interfaces related to the medical imaging platform are divided into three groups (see also section 2.9): interfaces within the generic skeleton (platform internal interfaces excluding the extensibility interfaces), interfaces towards the specific plug-ins (the extensibility interfaces), and interfaces towards the environment of the system (platform external interfaces). The difficulty to manage the evolution of the interfaces of these three groups increases in the order in which they are listed because of the growing size of the community that is involved.

6.6 Documentation

A component has its corresponding requirement and design documentation. These documents contain class and sequence diagrams specified with UML. Furthermore, special attention is paid to the various aspects as identified by the system architect, e.g. error handling or initialisation. The documentation of each component must address each aspect in a separate section.

The concept of adding a specific component to a generic component framework to realise specific behaviour also applies to the documentation. For both generic components and the specific plug-ins, separate documents are written. To find out the available functionality of a specific software component configuration, the corresponding documents can be viewed together.
The interfaces provided by components can be divided into three groups, viz. component specific interfaces, interfaces that are defined by a component framework that must be implemented by its plug-ins, and interfaces that are prescribed by the infrastructure and that relevant for a large number of units in the system (e.g. for initialisation purposes). The component developer is responsible for specifying the component specific interfaces. During specification and review, two other parties are involved: a member of the architecture team to guard the conceptual integrity, and the various clients of the interfaces. In case of an interface between the generic skeleton and specific parts, members of both the platform and product groups are involved, since the ownership lies with the platform group, but the product groups must use the interfaces.

6.7 Architecture Verification

For the medical imaging product family, architecture verification is being applied, as described in 0. Especially for this family, architecture verification is important, namely for the following reasons:

- The product family is very software intensive and complex. To manage complexity, the implicit architecture must meet the specified architecture to be able to manage the complexity.
- The product family will have a long lifetime, approximately 15 to 20 years. During this time, maintenance must be performed, new features must be added, etc. This is only possible when the architectural concepts are applied consequently throughout the system.
- Finally, work on the product family is performed at multiple distributed development sites. It is important that everywhere the same architectural concepts are being applied, to support communication between groups. This is especially of importance when, for example, a component developed by an application group is integrated with the platform.

The architect of the product family has specified a number of architectural rules that are checked automatically. Some of these rule have a global scope, others a regional scope. Examples of the checked rules are:

- The usage relations between units falls within the allowed usage relations as specified by the architect.
- All units must use certain infrastructure interfaces.
- Application units must provide an application service interface.
- Field-service interfaces, which are provided by various units, may only be used by the field-service unit.

7. EXPERIENCES IN THE CONSUMER DOMAIN

7.1 Product Characteristics

Consumer Electronics (CE) products such as televisions, set-top boxes and digital versatile disk (DVD) players embed a rapidly increasing amount of software, following Moore's law closely. CE products are sold in high volumes at competitive prices, making the *bill of material* an important issue. Consequently, the products are often severely resource constrained, at least from a computing point of view. High product quality is required, since repair in the field – though not impossible – is not attractive for the image of the company.

Philips produces a large diversity of CE products. Televisions already form a product family with over 100 variants. CE products have many things in common but have also many differences. Some members are even almost completely disjoint (e.g. a CD player and a television). This adds another dimension to product families – we'd rather speak of *product populations*.

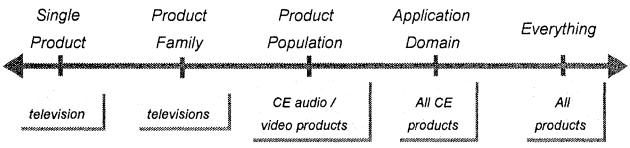


Figure 12: The domain hierarchy for CE products

Philips products are developed by multiple business groups at various locations all over the world, making software development inherently distributed. While the size of the software increases rapidly, the development time must decrease significantly to follow market trends quickly. Therefore, a substantial amount of reuse is required. We feel that only a compositional approach using 'subsystem platforms' can implement this effectively.

7.2 Software Components

Let's start with our foundation: a lightweight component model called *Koala* [30] inheriting from Microsoft's Component Object Model (COM) with an implementation scheme that induces no extra overhead (code size and performance) as compared with traditional (non-component) programming. Explicit design goal was to enable evolution to COM when our products become less resource constrained.

Other inspiration sources were Visual Basic and hardware ICs. As in COM, we define *interfaces* as small groups of functions, and model variation with the absence or presence (design-time or run-time) of such interfaces. As in Visual Basic, we have an explicit notion of *glue* code, as we do not believe that we can build a large variety of products by merely clicking components together. As in hardware, we make not only *provides* interfaces (outputs) explicit, but also all *requires* interfaces (inputs), and allow these to be explicitly bound by third parties to other components.

With respect to requires interfaces, COM components have many *implicit* context dependencies (e.g. the use of the Win32 platform), and only few *explicit* ones (such as connection points). Our approach (also inspired by [24]) forces developers to make *all* context dependencies explicit, allowing software architects to monitor and constraint these dependencies. Third party binding also enables the addition of glue code for e.g. monitoring calls or adding simple functionality. Our requires interfaces are in fact an example of variation points [17].

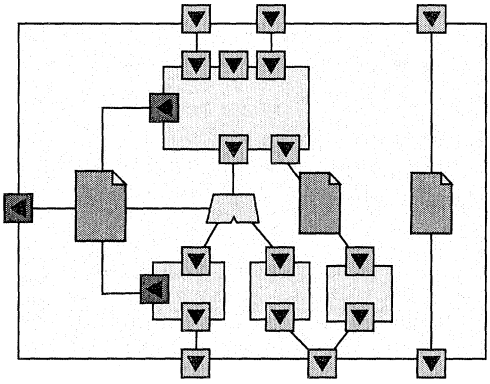


Figure 13: An example software component

The component model is essentially recursive. Compound components contain other components and define their mutual binding. Such compound components can have interfaces themselves; without external interfaces, they

are called 'configurations'. Interfaces are defined in an interface description language (IDL), components in a component description language (CDL). Figure 13 graphically illustrates our CDL, where components are represented by rectangles, interfaces by squares with embedded triangles, and glue code by 'documents' and 'trousers'.

7.3 Handling Diversity

The Koala model supports diversity by parameterisation (explicit and implicit) and by allowing structural variation.

A component can be *parameterised* with a large number of parameters (cf. properties in Visual Basic), grouped into *diversity interfaces* (usually drawn at the left side of components). The parameters (or functions) in such interfaces are assigned values in glue modules. These values may be expressions that use parameters or functions of other (diversity) interfaces of subcomponents (we call this the *diversity spreadsheet*).

Structural variation can be obtained by creating different compound components containing similar sets of subcomponents connected in different ways. We support the selection of alternative components (a form of semi-dynamic binding) with *switches* that parameterise the binding between components. Our model does not support plug-ins, but it does allow for '*plug-ons*' (components plugged into the border of a component).

We distinguish between *optional* and *mandatory* interfaces. Optional requires interfaces need not be connected; components must query for the presence of an implementation behind them. This is in fact an implicit parameterisation of the component. Components can assume that all mandatory requires interfaces are implemented without the need for a query.

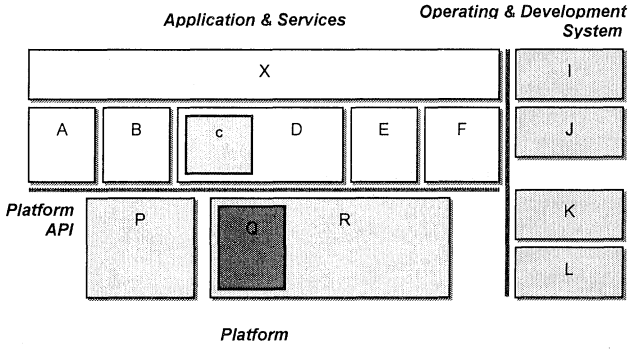


Figure 14: Layers and subsystems

7.4 Architectural Concepts

We divide our software into three *layers* (see Figure 14): the computing platform, the audio/video platform and 'services and applications'. Each of these layers has an independent evolution, driven by separate forces (innovation in computing hardware and operating systems, in signal processing hard- and software, and in user interface concepts). In addition, the capabilities for developing software in each of these layers are substantially different.

We further divide the software into *subsystems*. Subsystems implement subsystem domains as introduced in section 4.1. They also have independent evolution and require specific capabilities. A subsystem is technically a compound Koala component (but see below). Subsystems form large units of reuse within the product population.

Subsystems are composed of smaller components. These form the units of reuse *within* a subsystem domain; i.e. they can be used to create different variants of subsystems. A basic component consists of one or more *modules* (a pair of C and H files), and is typically implemented by a single developer.

Actually, our term *subsystem* is overloaded, as we use it both for a single compound component as part of a single product, and as a *package* of components and interfaces. Such a package adds a notion of scope to our model, as it can contain private and public interfaces and components. A package is really a unit of development, while a subsystem is a unit of deployment. We usually mean *package* when we use the term *subsystem*.

Subsystem diversity is implemented by offering different compound components in a package (i.e. different combinations of basic components), providing similar (but not equal) functionality in a subsystem domain. Each compound component can still be parameterised. Packages sometimes also contain small public components that can serve as reusable glue between subsystems.

7.5 A Common Platform Approach

Component technology is one part of the solution for creating a large variety of products; a platform approach is the other part. Our platform consists of a global architecture plus a set of reusable subsystems. Products can be created by combining subsystems. There is no sharing of software between products other than through the reusable subsystems! Reuse of software that first emerges in a product is obtained by integrating or promoting the software into a subsystem.

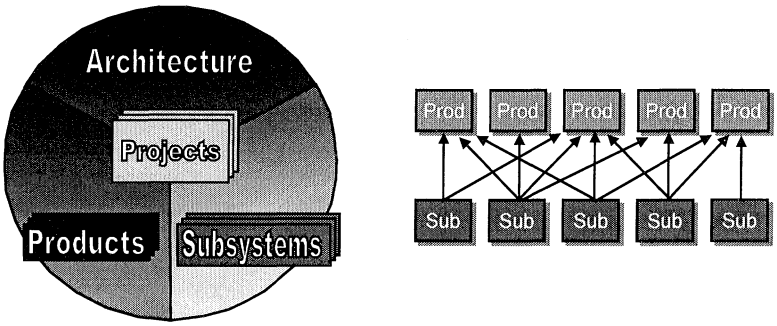


Figure 15: The three types of processes, and the subsystem product relation

Three types of processes are defined: (1) definition and evolution of *architecture*, (2) definition, implementation and evolution of *subsystems*, and (3) development of *products*. At each moment in time, there is precisely one process of type (1), but there are many instances of the second and third types. Many subsystems and products are developed in parallel; moreover, they usually have independent life cycles!

It is interesting to note a difference between the consumer and the medical domain here. Whereas in the medical domain, all subsystems (units) are developed in synchronisation and integrated into a single platform that is released to product developers, in the consumer domain subsystems are developed and released independently from each other. In the 'medical' approach, the responsibility for the integration and testing of the subsystems lies with the platform team at the expense of a slower release cycle. In the 'consumer' approach, release cycles can be much quicker (i.e. directly between subsystem team and product team), but the responsibility for integration and testing shifts to the product teams.

7.6 Architecture

We make a distinction between global and regional software architecture. The *global architecture* identifies the subsystems and it defines the code and document architecture, but it only chooses a few concepts and styles. All other concepts and styles are defined at the *regional* level. A region is either a single subsystem or a set of closely related subsystems. For example, definition of the interfaces of a subsystem is handled at the regional level, although essential interfaces are sometimes identified at the global level.

The reason for minimising global architectural decisions stems from our large variety of products. We cannot choose all concepts, styles and mechanisms globally: different types of products may need different choices.

We have to be very careful when making global choices, balancing the advantage of uniformity against the disadvantage of loosing flexibility.

The code architecture is an example of a global choice, dictated by our component model. Had we used the binary compatibility of COM as our technology, this would not have been necessary.

7.7 Process and Organisation

Although our total development is intrinsically multi-site, we require that each subsystem and each product be developed at one site. Communication between development sites is far from optimal: communication facilities have improved considerably over the past years (e-mail, video conferencing) but time (and cultural) differences between the continents are still dominant.

Architecture often concerns making compromises (e.g. within technical and organisational constraints). When identifying subsystems, we'd rather compromise on subsystem boundaries than allowing a subsystem to be developed at more than one site. Note that in the end, the organisation should be adapted to the technical architecture, not vice versa. This can be done by defining capability centres that correspond with the sub-product domains.

A common choice for multi-site product development is to install a single distributed configuration management system. We deliberately choose not to do this. Each subsystem project has its own local CM system, and releases versions of the subsystem by publishing ZIP files on the Intranet. Source code is fully available, thus promoting an open source community.

7.8 Implementing Diversity

Building product families is all about making the right choices at the right time. We distinguish the following *decision moments*: component design, subsystem design, product design, factory, dealer, and customer.

Whatever a component designer *can* decide, he should build into his component, but product specific information he should not include. Instead, he should parameterise the component with such information. Similarly, a subsystem designer can bind certain parameters of sub-components while defining others in terms of subsystem parameters. The product designer performs the ultimate binding of parameters to constants or to values stored in a non-volatile memory. The factory initialises the NVM, but the dealer and the customer also get an opportunity to change the values.

Koala supports *late compile time binding*. Late binding is binding that occurs at product design time, at which time the compiler can still be run to generate optimal code for that product configuration.

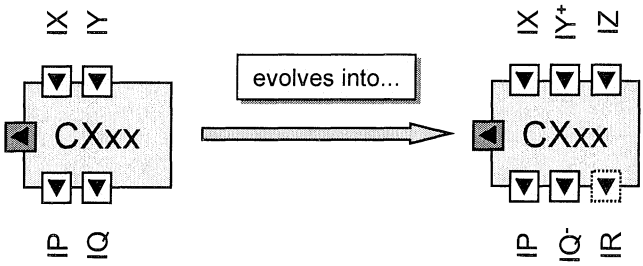


Figure 16: Possible evolution of a component

7.9 Managing Evolution

As in COM, our interfaces are immutable, but new interfaces can be defined. This rule is effected as soon as there are multiple users of the interface, in other words, if a global change is no longer feasible.

Our components are not immutable. New versions of components may be created, but must be upward compatible with older versions. This implies that they must provide all of the old functionality or more, and require all of the old functionality or less. So provides interfaces can be added or given a 'wider' type (subtypes may be connected to supertypes). Requires interfaces cannot be removed but given a narrower type or made optional.

Subsystems evolve by the evolution of the public interfaces and components of the subsystem. Interfaces and components can be deprecated but only if they are not used in any product anymore.

7.10 Documentation

The documentation of our asset base differs from the traditional software documentation (writing requirement specifications and design documents). We describe interfaces formally with IDL (syntax only) and informally with *interface data sheets* (syntax and semantics). These descriptions are independent from any implementation; our interfaces are reusable units of specification.

We describe components formally with CDL, and informally with *component data sheets*. The data sheet is only concerned with the external view; the internal view of a component is described in an *implementation notes* document. The component data sheet plays the role of a specification (written on beforehand) and a user manual (written afterwards).

7.11 Architecture Verification

CDL actually serves as an architectural description language (ADL). Having an ADL does not relieve us from architecture verification; it enables it! We envisage an architect's workbench that interactively performs consistency and evolution checks.

Consistency checks ensure that all configurations in the archive can be built. Remember that developers test components only in limited contexts - it is not realistic to test each change in all configurations that include the component. However, we can analyse the architecture for consequences of such changes. We can also detect patterns that are likely to be programming errors, such as binding a control interface without binding the corresponding notification interface.

Evolution checks test whether components are upward compatible with previous versions, an early warning mechanism for detecting potential build problems. Strong compatibility uses the rules as described above; weak compatibility allows violations, provided no existing configuration is bothered by the change.

8. DISCUSSION AND CONCLUSIONS

8.1 Applicability

In Section 1 we mentioned the existence of at least five different kinds of PCP's in the electronics industry. In this chapter, we presented the software renovation of two product families that are in their mature lifecycle phase. Several techniques and mechanisms were presented that are particularly suitable for products in this mature phase. Based on the different quality profiles for the two families a number of different choices were made with respect to for instance the component model part of the product family engineering approach. We have not described product family engineering approaches in other phases of Figure 1 in this chapter, but from our experiences with other product families in the other phases we know that also there significant variations in and on the main ingredients of the product family engineering approach exist. We incline therefore to make the tentative conclusion that the product family engineering approach presented is, in its turn, a kind of method family itself. The variations are now variations on the principles that were described in Section 2.

8.2 Product Family Engineering Principles

In Section 2 we presented a number of key principles: domain orientation, integral quality, components, architecture, domain solution capabilities, hardware abstraction, platforms, interfaces, and process integration. In Table 3, we sketch the commonality and variability across the two domains of these principles.

8.3 The Product Family Engineering Approach

In Section 3 we sketched a coarse framework for our product family engineering approach. The full approach, however, covers the Business, Organisation, Process and Architecture (BOPA) aspects of product family engineering. In this chapter, we only addressed the architectural (A) and the related development process (P) aspects. The Business (B) and organisation (O) were not dealt with explicitly. In the technical discussion, we concentrated on domain engineering and the architecting for product families. In practice, it turned out that the business and organisational issues are also very important.

Table 3: Comparison of principles across the two application domains

PRINCIPLE	MEDICAL	CONSUMER
Domain orientation	Inherently different	Inherently different
Integral quality	Different quality profiles, safety very important	Different quality profiles, cost dominant
Dependent independence	Industry standard component model	Customised cost optimised component model
Architecture centric	Strong architecture focus	Strong architecture focus
Uncoupling domain solution capabilities	Encapsulated in units	Encapsulated in subsystems
Hardware abstraction	Yes	Yes
Targeted value at low cost	Through shared platform and platform plug-ins	Through component configuration
Design for change	Explicit interfaces	Explicit interfaces
Co-operating in separation	Based on AFE, CSE, and ASE separation. Strong domain modelling	Based on AFE, CSE, and ASE separation. More implicit domain modelling

8.4 Domain Engineering

The concepts and techniques mentioned in Section 4 were used in both application domains. In the medical domain, however, much more explicit domain models were used than in the consumer domain. Again we saw interesting differences, for instance, the notion of variation points was realised in totally different ways, see Section 8.5. The domain modelling activity proved to be a good basis for the design of the family and the identification of the diversity.

8.5 Architecting for Product Families

The concepts and techniques mentioned in Section 5 were again used in both domains.

1. The medical family is not very resource constrained, so that a standard component technology can be used (COM). In CE, Koala was needed to meet these resource requirements.
2. The medical platform proved to be a really variance free generic architecture with plug-ins to handle diversity. The CE platform provides a set of building blocks from which different architectures can be instantiated. The medical platform is more pre-integrated than the CE platform, differing in releasing (parts of) the platform.
3. The medical platform aims at a product family. The CE platform aims at a product population, The medical platform defines more structure (skeleton) than the CE platform (family vs. population).
4. Different ways in supporting diversity were used (component frameworks, plug-ons).
5. For the medical platform, central integration and testing is an important requirement (FDA). For the CE platform, flexibility in creating multiple product architectures is the primary requirement. The medical platform is integrated and released at a single site and at a single point in time. The CE platform is released per subsystem.
6. Component frameworks provided an important means to realise diversity in the platform, and are suitable for multi-site development (component framework by platform group, and the plug-ins by the product groups)
7. A number of architectural principles have been successfully applied to support evolvability, viz. layering (incl. hardware abstraction) and dependent independence (separate units for subsystem domains).
8. Due to high safety and reliability requirements (FDA) shared testing of integrated platforms is better in the medical situation than completely

independent releases of the individual units. Automated architecture verification is in place to guard the conceptual integrity.

9. FURTHER RESEARCH

The notion of architectural verification is part of a bigger concern of platform guarding. Research into more advance platform guarding techniques is required. The notion of integral quality is very important. In practice, it is only feasible to treat a small number of qualities explicitly. More advanced approaches are needed. Two other technical topics that are subject of further research are family and platform evolution, addressing product derivation and traceability. The study of the organisational and business dimensions of the product family approach is also subject of further study. Last but not least, the notion of a method family for developing product families is intriguing and deserves further study.

ACKNOWLEDGEMENT

We want to thank our colleagues: Ben Pronk, Robert Deckers, Jaap van der Heijden, Auke Jilderda, Frank van der Linden, Jürgen Müller, Gerrit Muller, Henk te Sligte, Luc Koch and William van der Sterren for their contributions, stimulating discussions, or comments on the chapter.

This work has been partly conducted in the context of the ITEA 99005 project ESAPS as part of the Eureka Σ! 2023 Programme.

10. REFERENCES

1. P. America. *Requirements modeling for families of complex systems*. Submitted to the Third International Workshop on Software Architectures for Product Families, Las Palmas de Gran Canaria, Spain, March 15-17, 2000.
2. L. Bass, P. Clements, S. Cohen, L. Northrop, and J. Withey. *Product Line Practice Workshop Report*. CMU/SEI-97-TR-003.
3. L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 1998.
4. G. Booch, I. Jacobson, and J. Rumbaugh. *The Unified Modeling Language User Guide*. Addison-Wesley 1998.
5. F. Buschman, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *A System of Patterns*. Addison-Wesley, 1996.
6. P. Clements and L. Northrop. *A Framework for Software Product Line Practice*. version 2.0, 1999, SEI.
7. P. Coad and E. Yourdon. *Object-Oriented Analysis*. Yourdon Press/Prentice Hall,1990.

8. A.M. Davis. *201 principles of software development*. Mc Grawhill, 1994.
9. J.-M. DeBaud and K. Schmid. *A systematic approach to derive the scope of software product lines*. Proceedings ICSE 21, page 34-43, Los Angeles, 1999.
10. T. Dolan, R. Weterings, and J.C. Wortmann. *Stakeholders in Software-System Family Architectures*. Proceedings of the Second International ESPRIT ARES Workshop, F.J. van der Linden (Ed.), Springer LNCS 1429, pages 172-187, 1998.
11. B.P. Douglass. *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks and Patterns*. Addison-Wesley, 1999.
12. F. J. Erens. *The synthesis of variety*. PhD thesis Eindhoven University of Technology, ISBN 90-386-0295-6, 1996.
13. C. Hofmeister, R. Nord, and D. Soni. *Applied Software Architecture*. Addison-Wesley, 1999.
14. The draft *Recommended Practice for Architectural Description,* IEEE P1471/D51 of October 1999.
15. I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. ACM Press/Addison-Wesley, 1992.
16. I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1998.
17. I. Jacobson, M. Griss, and P. Jonsson. *Software Reuse – Architecture, Process, and Organization for Business Success*. Addison-Wesley, 1998.
18. E. Jandourek. *A Model for Platform Development*. Hewlett-Packard Journal, August 1996.
19. H. Kaindl. *Difficulties in the Transition from OO Analysis to Design*. IEEE Software, pages 94-102, September/October 1999.
20. E.-A. Karlsson (Ed.). *Software Reuse: A Holistic Approach*. Wiley, 1995.
21. P. Kruchten. *The 4+1 View Model of Architecture*. IEEE Software, pages 42-50, November 1995.
22. W.C. Lim. *Managing Software Reuse*. Prentice Hall, 1997.
23. F.J. van der Linden and J.K. Müller, *Creating Architectures with Building Blocks*. IEEE Software Vol. 12, No. 6, pages51-60, November 1995.
24. J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. *Specifying Distributed Software Architectures*. Proceedings ESEC'95, Wilhelm Schafer, Pere Botella (Eds.), Springer LNCS 989, pp. 137-153, 1995.
25. M.H. Meyer and A. Lehnerd. *The Power of Product Platforms: Building Value and Cost Leadership*. Free Press, ISBN 0-684-82580-5.
26. J.K. Müller. *Aspect Design with the Building Block Method*. Proceedings of the First Working IFIP Conference on Software Architecture, February 1999.
27. J.H. Obbink. *Product differentiation and Process Integration: the key to just-in-time in product development*, LNCS, 1995.
28. J.H. Obbink. *Analysis of Software Architectures in High- and Low-Volume Electronic Systems and industrial experience report*, LNCS, 1997.
29. J.H. Obbink, P.C. Clements, and F.J. van der Linden. *Introduction* of Proceedings of the Second International ESPRIT ARES Workshop, F.J. van der Linden (Ed.), Springer LNCS 1429, 1998.
30. R. van Ommering. *Koala. A Component Model for Consumer Electronics Product Software*. Proceedings of the Second International ESPRIT ARES Workshop, F.J. van der Linden (Ed.), Springer LNCS 1429, pages 76-86, 1998.
31. A. Postma, R.L. Krikhaar, and M. Stroucken. *A Method for Software Architecture Verification*. Submitted to ICSE 2000, Limerick, June 2000.

32. B. Pronk. *Medical Product Line Architectures - 12 years of experience*. Proceedings of the First Working IFIP Conference on Software Architecture, February 1999.
33. E. Rechtin and M.W. Maier. *The Art of Systems Architecting*. CRC Press, 1997, ISBN 0-8493-7836-2.
34. J. Rozenblit and K. Buchenrieder. *Codesign: Computer-aided software/Hardware Engineering*. IEEE Press, 1995, ISBN 0-7803-1049-7.
35. D. Soni, R. Nord, and C. Hofmeister. *Software Architecture in Industrial Applications*. Proceedings of the International Conference on Software Engineering, pages 196-210, Seattle, April 1995.
36. C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.
37. D.M. Weiss and C.T.R. Lai. *Software Product-Line Engineering: A Family Based Software Development Process*. Addison-Wesley, 1999.
38. J.G. Wijnstra. *Component Frameworks for a Medical Imaging Product Family*. Submitted to the Third International Workshop on Software Architectures for Product Families, Las Palmas de Gran Canaria, Spain, March 15-17, 2000.

Chapter 5

SYNTHESIS-BASED SOFTWARE ARCHITECTURE DESIGN

Bedir Tekinerdoğan and Mehmet Akşit
TRESE Group, Department of Computer Science, University of Twente, postbox 217, 7500 AE, Enschede, The Netherlands. email: {bedir, aksit}@cs.utwente.nl, www: http://trese.cs.utwente.nl

Keywords: Software architecture design, Synthesis, Domain analysis, Problem-Solving, Synbad

Abstract: Software architectures provide the gross-level design and as such impact the quality of the entire system. To support the quality factors such as robustness, adaptability and maintainability, a proper scoping of the architecture boundaries and likewise the identification of the relevant architectural abstractions is necessary. Several architecture design approaches have been introduced whereby the scoping of the architecture is merely based on the stakeholder's perspective. This chapter introduces a novel software architecture design approach that aims to scope the architecture boundaries from a systematic problem-solving perspective instead. In this so-called synthesis-based architecture design approach (Synbad), the client's perspective is abstracted to derive the technical problems. The technical problems define the scope of the solution domains from which the architectural abstractions are derived. The approach is illustrated for the design of an atomic transaction architecture for a real industrial project.

1. INTRODUCTION

Research on software architecture design approaches is still in its progressing phase and several architecture design approaches have been introduced in the last years [6][15][37][50]. However, a consensus on the appropriate software architecture design process is not established yet and

current software architecture design approaches have to cope with several problems[1].

First of all, planning the architecture design phase is intrinsically difficult due to its conflicting goals of providing a gross level structure of the system and at the same time directing the subsequent phases in the project. The first goal requires planning the architecture in later phases of the software development process when more information is available. In contrast, the latter goal requires planning it as early as possible so that the project can be more easily managed.

Second, most software architecture design approaches derive the architectural abstractions merely from the client's-perspective[2] rather than on the architectural solution perspective of the system. The gap between the client perspective and the architectural design perspective, however, is generally too large and the client may lack to specify the right detail of the problem. Due to the inappropriate scoping of the problem the fundamental transparent abstractions may be missed and/or redundant abstractions may be elicited.

Third, the adopted sources from the client's perspective are not very useful in providing sufficiently rich semantics of the architectural components and in providing guidelines for composing the architectural abstractions. In this case, architectural components are often equivalent to semantically poor groupings.

Finally, although solution domain analysis may be used and be effective in deriving the architectural abstractions and provide the necessary semantics, it may not suffice if it is not managed well. The problem is that the domain model may lack the right detail of abstraction to be of practical use for deriving architectural abstractions.

Current architecture design approaches have to cope with one or more of the above problems. In this chapter, a novel approach termed *synthesis-based software architecture design, Synbad* for short, is proposed, which aims providing effective solutions to these problems. In this approach the *synthesis* concept of traditional engineering disciplines is applied to the software architecture design process. Hereby, the requirements are first mapped to technical problems. For each problem the corresponding solution domain is identified and architectural abstractions are derived from the solution domain knowledge. Finally, the individual sub-solutions are synthesized in the overall software architecture. The novelty of this approach

[1] Chapter 1 of this book provides an in-depth analysis of the current architectural design methods.

[2] We use the term client to denote any stakeholder who has interest in the application of a software architecture.

is that it explicitly integrates the processes of technical problem analysis, solution domain analysis and alternative design space analysis.

The approach will be demonstrated using a project on the design of an atomic transaction system architecture for a distributed car dealer information system[3].

The remainder of the chapter is organized as follows. In section 2, the synthesis concept is described and a model for software architecture synthesis is derived. In section 3, an example project on the design of a software architecture for atomic transactions for a distributed car dealer information system will be described, which will be used throughout the whole chapter. In section 4, Synbad will be presented that will be illustrated for the example project. Finally, in section 5, we will present our discussion and conclusions.

2. SYNTHESIS

Software architecture design can be considered as a problem solving process in which the problem represents the requirement specification and the solution represents the software architecture design [35]. A well-known and widely applied problem solving technique in traditional engineering disciplines such as electrical engineering, chemical engineering and mechanical engineering is the concept of *synthesis* [37]. In section 2.1 we will explain this concept of *synthesis* as it is described in traditional engineering disciplines. In section 2.2 we will provide a software architecture synthesis model that represents the integration of the synthesis concept in software architecture design and as such forms a basis for Synbad, the synthesis-based software architecture design approach.

2.1 Synthesis in Traditional Engineering

Synthesis in engineering often means a process in which a problem specification is transformed to a solution by first decomposing the problem into loosely coupled sub-problems that are independently solved and integrated into an overall solution. In particular, the synthesis process includes an explicit phase for searching solution domains, searching design alternatives in the corresponding solution domain and selecting these alternatives based on explicit quality criteria.

[3] This work has been carried out as part of the INEDIS project that was a collaborative project between Siemens-Nixdorf and the TRESE group, Software Engineering, Dept. of Computer Science, University of Twente.

Synthesis consists generally of multiple steps or cycles. A synthesis cycle corresponds to a transition (transformation) from one *synthesis state* to another and can be formally defined as a tuple consisting of a problem specification state and a design state [38]. The problem specification state defines the set of problems that still needs to be solved. The design state represents the tentative design solution that has been lastly synthesized. Initially, the design state is empty and the problem specification state includes the initial requirements. After each synthesis state transformation, a sub-problem is solved. In addition a new sub-problem may be added to the problem specification state.

Each transformation process involves an evaluation step whereby the design solutions so far (design state) are evaluated with respect to its consistency with the initial requirements and any additional requirements identified during the synthesis.

A *synthesis-based design process* is defined as a finite sequence of synthesis states, resulting in a terminal state. A synthesis state is terminal in either of two cases: the specification part is satisfiable by the design part (there is a solution) or neither the design nor the specification can be modified. The first is a successful design the latter is an unsuccessful one.

The sub-solutions and overall solution has to meet a set of objective metrics, while satisfying a set of constraints. Constraints may be imposed within and among the sub-solutions. For a suitable synthesis it is required that the problem is understood well. This means that the problem is well-described and the quality criteria and constraints are known on beforehand. In practice, however, this is very difficult to meet and a complete analysis is impossible in any but the simplest problems [17]. Therefore, in practice, synthesis can usually start before the problem is totally understood.

During the synthesis process a designer needs to consider the design space that contains the knowledge that is used to develop the design solution. For this, synthesis requires the ability to produce a set of alternative solutions and select an optimal or near optimal solution. The space of possible solutions, however, may be very large and it is not feasible to examine all possible solutions [17].

In [38] it has been shown that the design synthesis is inherently an NP-complete problem. To manage this inherent complexity, synthesis can be performed at different, higher abstraction levels in the design process. In the design of digital signal processing systems, for example, the following synthesis approaches with increasing abstraction levels are distinguished: circuit synthesis, logic synthesis, register-transfer synthesis, and system synthesis [23]. For large problems, the lower-level design synthesis approaches become intractable and time consuming due to the large number of entities and their relations that need to be considered. In the example of

digital signal processing adopting the transistor as the basic abstraction, is unsuitable for current industrial problems that integrate millions of components. A higher level of abstraction reduces the number of entities that a designer has to consider which in turn reduces the complexity of the design of larger systems. In addition, higher level abstractions are closer to a designer's way of thinking and as such increases the understandability, which on its turn facilitates to consider various alternatives more easily. The counterpart is that higher level abstractions consist of the fixed configuration of lower level abstractions thereby implicitly reducing the alternative configuration possibilities. This is acceptable, though, since usually the total space of a synthesis from higher level abstractions is large enough to be of practical use.

2.2 Software Architecture Synthesis Model

Figure 1 represents a conceptual model for software architecture synthesis [35]. This model has been derived from our experiences in designing software architectures for various applications [3][5][63]67]. In addition it conforms to the synthesis-based design as it is widely accepted in mature engineering disciplines. Basically, it aims to explicitly integrate the processes of technical problem analysis, solution domain analysis, and alternative space analysis.

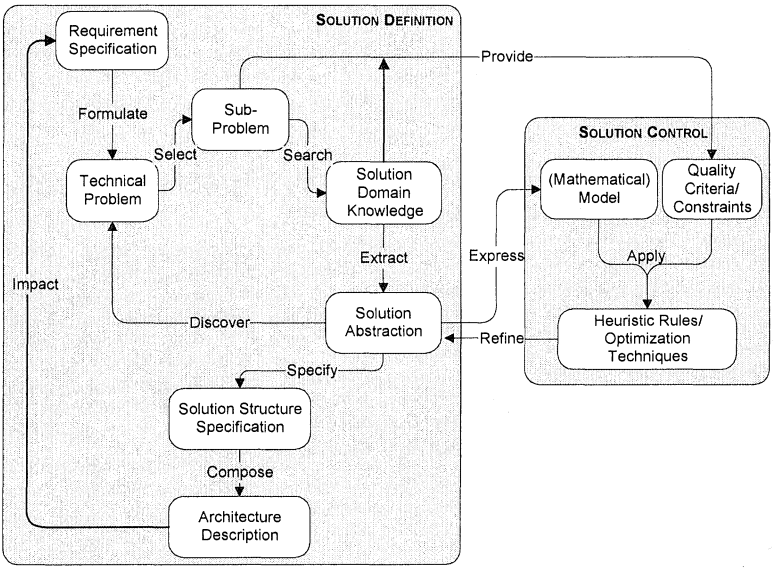


Figure 1: Architecture synthesis model

The model consists of two parts: *Solution Definition* and *Solution Control*. Each part consists of concepts and functions among concepts. The concepts are represented by rounded rectangles, the functions are represented by arrows. The part *Solution Definition* represents the identification and definition of solution abstractions. The part *Solution Control* represents the quantification, measurement, optimization and refinement of the selected solution abstractions. Note that this model represents a conceptual view of the software architecture synthesis process and does not enforce specific control flows between the various processes. In the following we will explain the concepts and functions of both parts of the model.

2.2.1 Solution Definition

The concept *Requirement Specification* represents the requirements of the stakeholders who are interested in the development of a software architecture.

The concept *Technical Problem* represents the problem specification that is actually to be solved. The model thus explicitly separates the concepts *Requirement Specification* and *Technical Problem*.

The function *Formulate* utilizes the *Requirment Specification* to define the process for searching and representing the problems that need to be solved for the architecture development.

The concept *Sub-Problem* represents a sub-problem of the identified problem.

The function *Select* represents the process for selecting the corresponding sub-problem from the problem.

The concept *Solution Domain Knowledge* represents the solution domain knowledge that is needed for solving the sub-problem.

The function *Search* represents the process for searching the solution domain knowledge for a given problem.

The concept *Solution Abstraction* represents the extracted solution from the solution domain knowledge.

The function *Extract* represents the process for extracting the solution abstractions from the solution domain knowledge.

The concept *Solution Structure Specification* represents the specification of the extracted solution abstraction.

The function *Discover* represents the process of discovering new sub-problems when new solution abstractions are extracted from the solution domain knowledge.

The function *Specify* represents the process for specifying the solution abstraction.

The concept *Architecture Description* represents the architecture description so far.

The function *Compose* represents the refinement of the overall-architecture description with the concept *Solution Structure Specification.*

The function *Impact* represents the process of refining the requirement specification from the results of the architecture specification.

2.2.2 Solution Control

The part *Solution Control* has conceptual relations with the part *Solution Definition* through the functions *Provide*, *Express* and *Refine*.

The function *Provide* represents the process for providing the quality criteria and constraints that are imposed on the solution. The concept *Quality Criteria/Constraints* represents these criteria and constraints of the (sub-) problem. These are derived from the technical problems and/or the solution domain knowledge.

The function *Express* represents a formalization of the solution abstraction for evaluation purposes. Typical formalizations may be the quantification into mathematical models.

The function *Apply* represents the process for measurement of the expressed solution abstraction using the provided quality criteria/constraints.

The concept *Heuristic Rules/Optimization Techniques* represents the optimization of the formalizations of the solution abstractions using the quality criteria and the constraints. It can be based on mathematical optimization techniques or heuristic rules.

The function *Refine* represents the process of refining the solution abstraction according to the results of the optimization techniques.

3. EXAMPLE PROJECT: TRANSACTION SOFTWARE ARCHITECTURE DESIGN

The Integrated New European Dealer Information System project (INEDIS) has been carried out as a collaborative project between the TRESE group of the University of Twente and Siemens-Nixdorf, The Netherlands. The project dealt with the development of a distributed car dealer information system in which different car dealers are connected through a network. A basic requirement was the automated support for processes such

as workshop processing, order processing, stock management, new and used car management, and financial accounting. Further, the car dealer system required the execution of the tasks consistently and effectively. The meaning of consistency, in general, depends on any a priori constraints, which must be guaranteed that they will never be violated. For example, two clients may not reserve the same car at the same time.

In a distributed system as the car dealer system, there are two main factors that threaten the consistency of data: *concurrency* and *failures*. In case of concurrency the executions of programs that access the same objects can interfere. When a failure occurs, one or more application programs may be interrupted in midstream. Since a program is written under the assumption that its effects are only correct if it would be executed in its entirety, an interrupted program may lead to inconsistencies as well. To achieve data consistency, distributed systems should include provision for both concurrency and recovery from failures. In addition it is generally demanded that the implementation of these concurrency and recovery mechanisms is transparent to the application program developers, since they will need only the primitives and do not want to be bothered with implementation details. *Atomic transactions*, or simply transactions, are a well-known and fundamental abstraction which provide the necessary concurrency control and recovery mechanisms for the application programs in a transparent way. Transactions relieve application programmers of the burden of considering the effects of concurrent access to objects or various kinds of failures during execution. Atomic transactions have proven to be useful for preserving the consistency in many applications like airline reservation systems, banking systems, office automation systems, database systems and operating systems.

The car dealer information system also required the use of atomic transactions, and was to be used in different countries and by different dealers each requiring dedicated transaction protocols. Therefore, a basic requirement of the system was to identify common patterns of transaction systems and likewise provide a stable architecture of atomic transactions that could be customized to the corresponding needs.

In addition to the need for adaptability at initialization time, the system required also adaptation at run-time. The car dealer system is constituted of a large number of applications with various characteristics, operates in heterogeneous environments, and may incorporate different data formats. To achieve optimal behavior, this requires transactions with dynamic adaptation of transaction behavior, optimized with respect to the application and environmental conditions and data formats. The adaptation policy, therefore, must be determined by the programmers, the operating system or the data

objects. Further, reusability of the software is considered as an important requirement to reduce development and maintenance costs.

4. SYNBAD: SYNTHESIS-BASED SOFTWARE ARCHITECTURE DESIGN PROCESS

In this section, the synthesis-based software architecture design process that implements the process of the Architecture Synthesis Model of Figure 1 will be described.

The following sections are organized around the basic processes of the approach. Section 4.1 describes the *Requirements Analysis process*, section 4.2 the *Problem Analysis process*, section 4.3 the *Solution Domain Analysis process*, section 4.4 *Alternative Space Analysis process* and finally section 4.5 the *Architecture Specification process*.

4.1 Requirements Analysis

The architecture design is initiated with the requirements analysis phase in which the basic goal is to understand the stakeholder requirements. Stakeholders may be managers, software developers, maintainers, end-users, customers etc. [28]. The requirements analysis process concerns the concept *Requirement Specification* of Figure 1.

In Synbad the well-known requirement analysis techniques such as informal requirement specifications, use-cases [21] and scenarios [32], constructing prototypes and defining finite state machine modeling are applied. Informal requirement specification serves as a first basis for the requirements analysis process and is generally defined by interacting with the clients. Use cases provide a more precise and broader perspective of the requirements by specifying the external behavior of the system from different user perspectives. Scenarios are instances of use cases and define the dynamic view and the possible evolution of the system. Prototypes are used to define the possible user interfaces and may further help to clarify the desired behavior of the system. Finally, for safety-critical systems rigorous approaches such as state transition diagrams or formal specification languages may be used.

These techniques have been applied in different approaches and have shown to be useful in supporting the analysis and understanding of the client requirements. We will not elaborate on these in this chapter and refer for detailed information to the corresponding publications [59][51][35].

4.2 Technical Problem Analysis

The requirements analysis process provides an understanding of the client perspective of the software system. In the technical problem analysis process the identified client requirements are mapped to technical problems. The underlying motivation for this technical problem analysis process is the idea that software architecture is in essence a problem solving process in which the solution represents an architecture design. In this sense, the technical problem analysis process is necessary to identify the essence of the problem, separate from the client's view on the problem. In the ideal case the client's view may represent directly the problem of concern, though, in practice this is far from truth and additional steps are required to capture the real problems.

The technical problem analysis process is related to the concepts *Technical Problem* and *Sub-Problem* and the functions *Select*, *Search* and *Discover* in the model of Figure 1. It consists of the following steps:

1. Generalizing the requirements: whereby the requirements are abstracted and generalized.
2. Identification of the sub-problems: whereby technical problems are identified from the generalized requirements.
3. Specification of the sub-problems: whereby the overall technical problem is decomposed into sub-problems.
4. Prioritization of the sub-problems: whereby the identified technical problems are prioritized before they are processed.

In the following we explain these processes in more detail.

4.2.1 Generalizing the requirements

Discovering the problems from a requirement specification is not a straightforward task. The reason for this is that the clients may not be able to accurately describe the initial state and the desired goals of the system. The client requirements may be specific and provide only specific interpretations of a more general problem. Therefore, to provide the broader view and identify the right problems we abstract and generalize from the requirement specification and try to solve the problem at that level[4]. Often, this abstraction and generalization process allows to define the client's wishes in

[4] In mathematics, solving a concrete problem by first solving a more general problem is termed as the *Inventor's Paradox* [44] [34]. The paradox refers to the fact that a general problem has paradoxically a simpler solution than the concrete problem.

entirely different terms and therefore may suggest and help to discover problems that were not thought of in the initial requirements.

4.2.2 Identification of the sub-problems

Once the requirement specification has been put into a more general and broader form, we derive the technical problem that consists usually of several sub-problems. At this phase, architecture design is considered as a problem solving process. Problem solving is defined as the operation of a process by which the transformation from the initial state to the goal is achieved [40]. We need thus first to discover and describe the problem. Therefore, in the generalized requirement specification we look for the important aspects that needs to be considered in the software architecture design [58]. These aspects are identified by considering the terms in the generalized requirements specification, the general knowledge of the software architect and the interaction with the clients. This process is supported by the results of the requirements analysis phase and utilizes the provided use-case models, scenarios, prototypes and formal requirements models.

4.2.3 Specification of the sub-problems

The identification of a sub-problem goes in parallel with its specification. The major distinction between the identification and the specification of a problem is that the first activity focuses on the process for finding the relevant problems, whereas the second activity is concerned with its accurate formalization. A problem is defined as the distance between the initial state and the goal. Thereby, the specification of the technical problems consists of describing its name, its initial state and its goal.

4.2.4 Prioritization of the sub-problems

After the decomposition of the problem into several sub-problems the process for solving each of the sub-problems can be started. The selection and ordering in which the sub-problems are solved, though, may have an impact on the final solution. Therefore, it is necessary to prioritize and order the sub-problems and handle the sub-problems according to the priority degrees. The prioritization of the sub-problems may be defined by the client or the solution domain itself. The latter may be the case if a sub-problem can only be solved after a solution for another sub-problem has been defined.

EXAMPLE

We generalized the INEDIS requirement specification [1] and mapped these to the technical problems. For example, we generalized the requirements for the various scheduling techniques. In the original requirement specification and the interview with the stakeholders we identified that only two concurrency control approaches were used, namely optimistic and aggressive locking. Attempts were made to adapt between these two concurrency control mechanisms. After our discussion with the stakeholders [55] it followed that the system needed also other types of concurrency control protocols and the run-time adaptation had to be defined for these as well. In parallel with our generalization of the requirements we were able to define the different sub-problems, which are listed in the following:

P1. Provide transparent concurrency control.
Goal: Determine the set of concurrency control techniques that are required and provide this in a reusable form.
P2. Provide transparent recovery techniques.
Goal: Determine the set of recovery techniques that can be used for various kinds of data types and provide this in a reusable form.
P3. Provide transparent transaction management techniques.
Goal: Provide various transaction management techniques that can be applied for advanced transactions such as long transactions and nested transactions. Provide the various start, commit and abort protocols in a reusable format.
P4. Provide adaptable transaction protocols based on transaction, system and data criteria.
Goal: Provide the means to adapt the transaction protocols both on compile-time and run-time. Adaptation mechanism should be determined by programmers, operating system or the data object characteristics.

4.3 Solution Domain Analysis .

The Solution Domain Analysis process aims to provide a solution domain model that will be utilized to extract the architecture design solution. It relates basically to the concepts *Solution Domain Knowledge* and *Solution Abstraction* and the functions *Search* and *Extract* in the model of Figure 1. The solution domain analysis process consists of the following activities:

1. Identification and prioritization of the solution domains for each sub-problem.
2. Identification and prioritization of the knowledge sources for each solution domain.
3. Extracting solution domain concepts from solution domain knowledge.
4. Structuring the solution domain concepts.
5. Refining the solution domain concepts.

In the following we will explain these steps in more detail.

4.3.1 Identification and prioritization of the solution domains

For the overall problem and each sub-problem we search for the solution domains that provide the solution abstractions to solve the technical problem. The solution domains for the overall problem are more general than the solution domains for the sub-problems. Further, each sub-problem may be recursively structured into sub-problems requiring more concrete solution domains on their turn.

An obstacle in the search for solution domains may be the possibly large space of solution domains leading to a time-consuming search process. To support this process, we look for categorizations of the solution domain knowledge into smaller sub-domains. There are different categorization possibilities [24]. In library science, for example, the categories are represented by *facets* that are groupings of related terms that have been derived from a sample of selected titles [48]. In [2], the solution domain knowledge is categorized into *application*, *mathematical* and *computer science* domain knowledge. The application domain knowledge refers to the solution domain knowledge that defines the nature of the application, such as reservation applications, banking applications, control systems etc. Mathematical solution domain knowledge refers to mathematical knowledge such as logic, quantification and calculation techniques, optimization techniques, etc. Computer science domain refers to knowledge on the computer science solution abstractions, such as programming languages, operating systems, databases, analysis and design methods etc. This type of knowledge has been recently compiled in the so-called Software Engineering Body of Knowledge (SWEBOK) [14]. Notice that our approach does not favor a particular categorization of the solution domain knowledge and likewise other classifications besides of the above two approaches may be equally used.

If the solution domains have been adequately organized one may still encounter several problems and the solution domain analysis may not always warrant a feasible solution domain model. This is especially the case if the

solution domains are not existing or the concepts in the solution domain are not fully explored yet and/or compiled in a reusable format.

If the solution domain knowledge is not existing, one can either terminate the feasibility analysis process or initiate a scientific research to explore and formalize the concepts of the required solution domain. The first case leads to the conclusion that the problem is actually not (completely) solvable due to lack of knowledge. The latter case is the more long-term and difficult option and falls outside the project scope.

If a suitable solution domain is existing and sufficiently specified, it can be (re)used to extract the necessary knowledge and apply this for the architecture development. It may also happen that the solution domain concepts are well-known but not formalized [30]. In that case it is necessary to specify the solution domain.

4.3.2 Identification and prioritization of knowledge sources

Each identified solution domain may cover a wide range of solution domain knowledge sources that represent the content of the related knowledge. These knowledge sources may not all be suitable and vary in quality. For distinguishing and validating the solution domain knowledge sources we basically consider the quality factors of *objectivity* and *relevance*. The objectivity quality factor refers to the solution domain knowledge sources itself, and defines the general acceptance of the knowledge source. Solution domain knowledge that is based on a consensus on a community of experts has a higher objectivity degree than solution domain knowledge that is just under development. The relevance qualoty factor refers to the relevance of the solution domain knowledge for solving the identified technical problem.

The relevance of the solution domain knowledge is different from the objectivity quality. A solution domain knowledge entity may have a high degree of objective quality because it is very precisely defined and supported by a community of experts, though, it may not be relevant for solving the identified problem because it addresses different concerns. To be suitable for solving a problem it is required that the solution domain knowledge is both objective and relevant. Therefore, the identified solution domain knowledge is prioritized according to their objectivity and relevancy factors. This can be expressed in the empirical formula [2]:

$$priority(s) = f(objectivity(s), (relevance(s))$$

Hereby *priority(), f(), objectivity()* and *relevance()* represent functions that define the corresponding quality factors of the argument *s*, that stands for solution domain knowledge source. For solving the problem, first the

solution domain knowledge with the higher priorities is utilized. The measure of the objectivity degree can be determined from general knowledge and experiences. The measure for the relevance factor can be determined by considering whether the identified solution domain source matches the goal of the problem. Note, however, that this formula should not be interpreted too strictly and rather be considered as an intuitive and practical aid for prioritizing the identified solution domain knowledge sources rather.

EXAMPLE

Let us now consider the identification and the prioritization of the solution domains for the given project example. For the overall problem, a solution is provided by the solution domain *Atomic Transactions*. Table 1 provides the solution domains for every sub-problem.

Table 1: The solution domains for the sub-problems

SUB-PROBLEM	SOLUTION DOMAIN
P1	*Transaction Management*
P2	*Concurrency Control*
P3	*Recovery*
P4	*Adaptability*

The prioritization of these solution domains was defined in the above order from P1 to P4.

For the overall problem and the corresponding solution domain of *Atomic Transactions*, we could find sufficient knowledge sources. Our identified solution domain knowledge sources consisted of managers, system developers, maintainers, literature on transactions, and documentation on the existing car dealer system. However, among these different knowledge sources we assigned higher priority values to the literature on atomic transaction systems. Table 2 provides the selected set of knowledge sources for the overall solution domain.

Table 2: A selected set of the identified knowledge sources for the overall solution domain

ID	KNOWLEDGE SOURCE	FORM
KS1	*Concurrency Control & Recovery in Database Systems* [11]	textbook
KS2	*Atomic Transactions* [36]	textbook
KS3	*An Introduction to Database Systems* [13]	textbook
KS4	*Database Transaction Models for Advanced Applications* [20]	textbook
KS5	*The design and implementation of a distributed transaction system based on atomic data types* [68]	journal paper
KS6	*Transaction processing: concepts and techniques* [26]	textbook
KS7	*Principles of Transaction Processing* [10]	textbook
KS8	*Transactions and Consistency in Distributed Database Systems* [61]	journal paper

The table consists of three columns that are labeled as *ID*, *Knowledge Source* and *Form* that respectively represent the unique identifications of the knowledge sources, the title of the knowledge source and the representation format of the knowledge source. The table includes the knowledge sources that describe atomic transactions in a general way. Knowledge sources that deal with specific aspects of transaction systems, for example such as deadlock detection mechanisms, have been temporarily omitted and are identified when the corresponding sub-problems are considered.

In the same manner we looked for knowledge sources for the individual sub-problems and we were able to identify many knowledge sources for the solution domains *Transaction Management*, *Concurrency Control* and *Recovery*. The solution domain *Adaptability* was more difficult to grasp than the other ones. For this, we did a thorough analysis on the notion of adaptability and studied various possibly related publications such as control theory [47][22][62]. In addition we organized a workshop on Adaptability in Object-Oriented Software Development [57][4].

As an example, Table 3 shows a selected set of the identified knowledge sources for the solution domain *Concurrency Control*.

Table 3: A set of knowledge sources for the solution domain *Concurrency Control*

ID	KNOWLEDGE SOURCE	FORM
KS1	*Concurrency Control in Advanced Database Applications* [7]	journal paper
KS2	*Concurrency Control in Distributed Database Systems* [16]	textbook
KS3	*The theory of Database Concurrency Control* [41].	textbook
KS4	*Concurrency Control & Recovery in Database Systems* [11]	textbook
KS5	*Concurrency Control and Reliability in Distributed Systems* [12]	journal paper
KS6	*Concurrency Control in Distributed Database Systems* [9]	textbook

Note that the knowledge source KS4 has also been utilized for the overall solution domain. The reason for this is that this knowledge source is both sufficiently abstract to be suitable for the overall solution domain and also provides detailed information on the solution domain *Concurrency Control*.

4.3.3 Extracting Solution Domain Concepts from Solution Domain Knowledge

Once the solution domains have been identified and prioritized, the knowledge acquisition from the solution domain sources can be initiated. The solution domain knowledge may include a lot of knowledge that is covered by books, research papers, case studies, reference manuals, existing prototypes/systems etc. Due to the large size of the solution domain knowledge, the knowledge acquisition process can be a labor-intensive activity and as such a systematic approach for knowledge acquisition is required [43], [25], [66].

In our approach we basically distinguish between the *knowledge elicitation* and *concept formation* process. Knowledge elicitation focuses on extracting the knowledge and verifying the correctness and consistency of the extracted data. Hereby, the irrelevant data is disregarded and the relevant data is provided as input for the concept formation process. Knowledge elicitation techniques have been described in several publications and its role in the knowledge acquisition process is reasonably well-understood [66], [39], [19], [21].

The concept formation process utilizes and abstracts from the collected knowledge to form concepts[5]. In the literature, several concept formation

[5] There are basically three views of concepts, including the classical view, the prototype view and the exemplar view. Concept forming through abstraction from instances is basically applied in the classical view and the prototype view [23].

techniques have been identified[6] [26][46][23]. One of the basic abstraction techniques in forming concepts is by identifying the variations and commonalities of extracted information from the knowledge sources [52][20]. Usually a concept is defined as a representation that describes the common properties of a set of instances and is identified through its name.

EXAMPLE

We analyzed and studied the identified solution domain knowledge according to the assigned priorities and extracted the fundamental concepts. After considering the commonalities and variabilities of the extracted information from the solution domains we could extract the following solution domain concepts [35]: *Atomic Transaction System, Transaction, TransactionManager, PolicyManager, Scheduler, RecoveryManager, DataManager, Data Object.*

4.3.4 Structuring the Solution Domain Concepts

The identified solution domain concepts are structured using *generalization-specialization* relations and *part-whole* relations, respectively. In addition, also other structural *association* relations are used. Like the concepts themselves, the structural relations between the concepts are also derived from the solution domains.

For the structuring and representation of concepts, so-called *concept graphs* are used. A *concept graph* is a graph which nodes represent *concepts* and the edges between the nodes represent *conceptual relations*. The notation of concept graphs is given in Figure 2.

The notation for a concept is a stereotype of the class notation in the Unified Modeling Language [13]. A stereotype represents a subclass of a modeling element with the same form but with a different intent. The stereotype for a concept is identified by the keyword <concept>[7].

[6] This process of concept abstraction is usually considered as a psychological activity that is often associated with the term 'experience' [52]. Experts, i.e. persons with lots of experience, own a larger set of concepts and are better in forming concepts than persons who lack this experience.

[7] Note that a class does not need to be similar to a concept. Although both classes and concepts are generally formed through an abstraction process this does not imply that every abstraction is a concept. A concept is a well-defined and stable abstraction in a given domain. Although the notation that we use for representing concepts is similar to the notation of classes, one should be aware that concepts are at a different abstraction level than classes and should be treated as such.

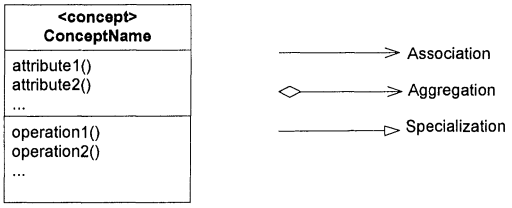


Figure 2: Notation for concept graphs

EXAMPLE

Figure 3 shows the structuring of the solution domain concepts in the top-level concept graph of transaction systems. The concept *Transaction Manager* has an association relation *manages* with the concept *Transaction*. This means that *Transaction Manager* is responsible for the atomic execution of *Transaction*. The association relation *manages* between concept *DataManager* and *Data Object* represents the maintenance of the consistency of data objects. Hereby, *DataManager* utilizes and coordinates the concepts *Scheduler* and *RecoveryManager* by means of the association relation *coordinates*. The concept *PolicyManager* coordinates the activities of the concepts *TransactionManager* and *DataManager* and defines the policy for adapting to different transaction protocols. Finally, the association relation *accesses* between *Transaction* and *Data Object* defines a read/update relation between these two. A more detailed descrption of these concepts is given in [35].

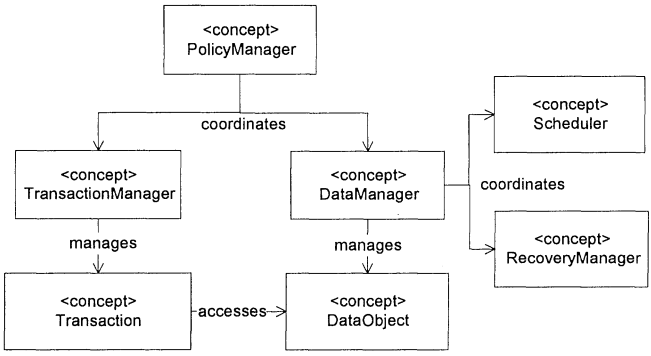


Figure 3: The top-level concept graph of an atomic transaction system

4.3.5 Refinement of Solution Domain Concepts

After identifying the top-level conceptual architecture we focus on each sub-problem and essentially follow the same synthesis process. The refinement becomes necessary if the architectural concepts have a complex structure themselves and this structure is of importance for the eventual system.

The ordering of the refinement process is determined by the ordering of the problems with respect to their previously determined priorities. Architectural concepts that represent problems with higher priorities are handled first. Due to space limitations we will not elaborate on the refinement of these concepts in this chapter but suffice to refer to [35] in which this is described in detail.

4.4 Alternative Design Space Analysis

We define the *alternative space* as the set of possible design solutions that can be derived from a given conceptual software architecture. The *Alternative Design Space Analysis* aims to depict this space and consists of the sub-processes *Defining the Alternatives for each Concept* and *Describing the Constraints*. Let us now explain these sub-processes in more detail.

4.4.1 Defining the Alternatives for each Concept

In Synbad, the various architecture design alternatives are largely dealt with by deriving architectural abstractions from well-established concepts in the solution domain. Each architectural concept is an abstraction from a set of instantiations and during the analysis and design phases the architecture is realized by selecting particular instances of the architectural concepts. An instance of a concept is considered as an alternative of that concept. The total set of alternatives per concept may be too large and/or not relevant for solving the identified problems. Therefore, to define the boundaries of the architecture it is necessary to identify the relevant alternatives and omit the irrelevant ones.

The alternatives of a given concept may be explicitly identified and published. In that case, selecting alternatives for a concept is rather straightforward and depends only on the solution domain analysis process. If the concepts have complex structures consisting of sub-concepts then an alternative is defined as a composition of instances of separate sub-concepts. The set of alternatives may then be too large to provide a name for each of them individually. Nevertheless, we need to depict the total set of alternatives so that each of them can be derived if necessary. For this, first

the alternatives of each sub-concept are identified, and consequently the various compositions of these alternatives are considered.

EXAMPLE

Let us now consider the alternatives for the concepts in the top-level architecture. We depict the alternative space by providing a table in which the column headers represent the sub-concepts and each table entry represents an instance of the sub-concept in the column header. For example, Table 4 represents the alternative space for the concept *Scheduler.* The table has 4 columns, the first one represents the numbering of alternatives and the second to the fourth columns represents the sub-concepts of the concept *Scheduler*[8].

Table 4: Alternatives of the sub-concepts of *Scheduler*

	A. SYNCHRONIZATION SCHEME	B. SYNCHRONIZATION STRATEGY	C. PERFORMANCE FAILURE DETECTOR
1.	Two Phase Locking	Aggressive	Deadlock Detector
2.	Timestamp Ordering	Conservative	Infinite Blocking Detector
3.	Optimistic		Infinite Restart Detector
4.	Serial		Cyclic Restart Detector

An alternative of the concept *Scheduler* is a composition of selections of the alternatives of the sub-concepts. For instance, an alternative that may be derived from Table 4 is the tuple *(Two Phase Locking, Conservative, Deadlock Detector*) which represents a scheduler that uses aggressive two phase locking protocol whereby a conservative deadlock detection mechanism is used. Note that the concept *Scheduler* has 4x2x4=16 theoretically possible alternatives.

Another example is given in Table 5, which represents the alternative space for the concept *RecoveryManager*.

[8] These have been derived in the refinement process in which the synthesis process has been applied to define the sub-architecture of the concept *Scheduler* [35].

Table 5: Alternatives of the sub-concepts of RecoveryManager

	A. LOG MANAGER	B. FAILURE ATOMICITY SYNCHRONIZER	C. RESTARTING	D. CHECKPOINTING
1.	Operation Logging	Recoverable	Undo / Redo	Commit-Consistent
2.	Deferred-Update	Cascadeless	No-Undo / Redo	Cache-Consistent
3.	Update-In-Place	Strict	Undo / No-Redo	Fuzzy
4.			No-undo / No-redo	

An alternative of the concept *RecoveryManager* is the tuple *(Operation Logging, Strict, Undo-Redo, Commit-consistent)*, representing a *RecoveryManager* that applies *Operation Logging*, *Strict* executions, adopts *Undo-Redo* algorithm in case of restarts and a *Commit-Consistent* checkpointing mechanism for optimizing the restart procedure. The total number of theoretically possible alternatives of *RecoveryManager* is 4x3x4x3=144. If alternatives of *Scheduler* and *RecoveryManager* are composed then the number of the set of possible alternative compositions equals 16x144=2304 alternatives. Obviously, not all the alternative compositions are possible or required and it is worthwhile to eliminate these alternatives. This process is described in the following section.

4.4.2 Describing Constraints between Alternatives

An architecture consists of a set of concepts that together define a certain structure. An instantiation of an architecture is a composition of instantiations of concepts [2][35]. The instantiations of these various concepts may be combined in many different ways and likewise this may lead to a combinatorial explosion of possible solutions. Hereby, it is generally impossible to find an optimal solution under arbitrary constraints for an arbitrary set of concepts.

To manage the architecture design process and define the boundaries of the architecture it is important to adequately leverage the alternative space. Leveraging the alternative space means the reduction of the total alternative space to the relevant *alternative space*. A reduction in the space is defined by the solution domain itself that defines the *constraints* and as such the possible combination of alternatives. The possible alternative space can be further reduced by considering only the combinations of the instantiations that are relevant from the client's perspective and the problem perspective.

Constraints may be defined for the sub-concepts within a concept as well as among higher-level concepts. We first describe the constraints among the sub-concepts within a concept and later among the peer-concepts. We use the *Object Constraint Language* (OCL) [64] that is part of the UML to express the constraints over the various concepts.

Constraint identification is not only useful for reducing the alternative space but it may also help in defining the right architectural decomposition. The existence of many constraints between the architectural components provides a strong coupling and as such it may possibly indicate a wrong decomposition. This may result in a reconsideration of the identified architectural structure of each concept.

4.5 Architecture Specification

The Architecture Specification process consists of the two sub-processes Extracting Semantics of the Architecture and Defining Dynamic Behavior of the Architecture.

4.5.1 Extracting Semantics of the Architecture

To provide a more formal specification the semantics of each individual concept is extracted from the solution domain. As a format for writing a formal specification we use:

<operation><pre-condition><post-condition>

Hereby, <operation> represents the name of the operation of a concept. The part <pre-condition> describes the conditions and assumptions made about the values of the concept variables at the beginning of <operation>. The part <post-condition> describe what should be true about the values of the variables upon termination of <operation>. Note that this is just one particular way of specifying architectures. For the specification of transaction architectures this type of specification was appropriate, however, other applications may require different specification mechanisms.

4.5.2 Define Dynamic Behavior of the Architecture

The specifications of the architectural components are used to model the dynamic behavior of the architecture. For this purpose the so-called collaboration diagrams are utilized [13]. Collaboration diagrams show the structural organization of the components and the interaction among these components. The collaboration diagrams are derived from the pre-defined specifications of the architectural concepts.

EXAMPLE

The collaboration diagram for the transaction architecture is given in Figure 4.

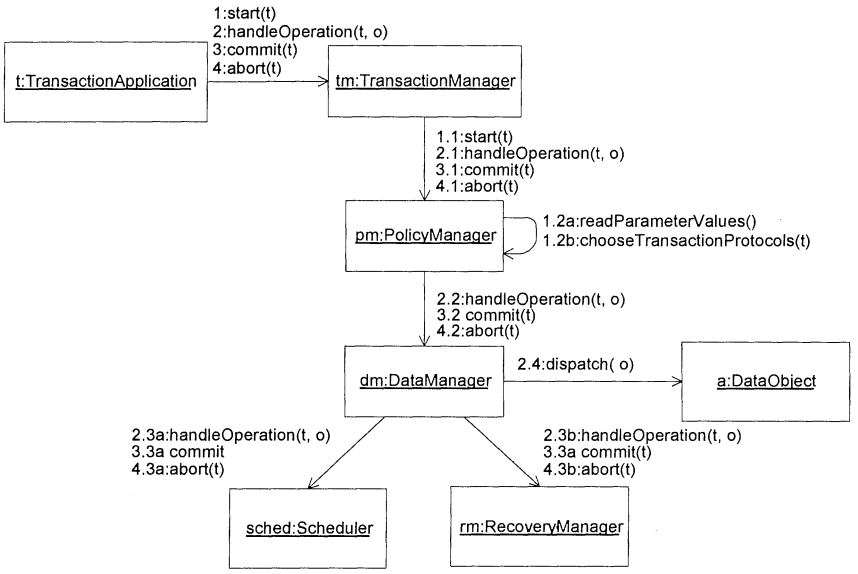


Figure 4: Collaboration diagram for the atomic transaction architecture

5. DISCUSSION AND CONCLUSIONS

In this chapter we have presented *Synbad*, the *synthesis-based software architecture design approach*[9]. This approach is based on the concept *synthesis* of mature engineering disciplines whereby the initial problem is decomposed into sub-problems that are solved separately and later integrated in the overall solution. The novelty of Synbad with respect to the existing architecture design approaches is that it makes the processes of *problem analysis*, *solution domain analysis* and *alternative space analysis* explicit. During the *problem analysis*, the client requirements are mapped to the technical problems providing a more objective and reliable description of the problem. During the *solution domain analysis*, stable architectural

[9] The web site of this approach is at: http://trese.cs.utwente.nl/architecture_design/

components with rich semantics are derived from the solution domain knowledge that includes well-defined and stable concepts. The solution domain analysis itself is leveraged by the pre-identified technical problems so that the right detail of the solution domain model is guaranteed. The *alternative space analysis* explicitly depicts the possible set of design alternatives that can be derived from the architectural components.

We have illustrated the approach by applying it for the design of an atomic transaction architecture for a distributed car dealer system in an industrial project. Apart from this, experimental studies have been carried out with earlier versions of this approach in pilot studies that were carried out by MSc students. For example, in [63], a software architecture for image algebra was derived for the laboratory for clinical and experimental image processing. The basic solution domain for this architecture was image algebra and several related publications could be identified from which sufficient stable abstractions were derived for the design of the software architecture. The atomic transaction and the image algebra domain appeared to be examples of well-defined and sufficiently formalized domains. The experimental studies have been, though, also applied on domains that are less formalized. In [5], for example, a software architecture has been derived for a Quality Management Systems for efficient information retrieval and in [67] a software architecture has been derived for insurance systems. In both cases, several publications could be identified on the corresponding domains, but in addition it was also necessary to refer to the factual knowledge and experiences for the design of the software architecture. The solution domain may thus consist of a combination of various forms of solution techniques such as theories, solution domain experts, and experiences in the corresponding domain.

In the following we will list the conclusions that we could obtain from our experience in applying Synbad to the project on atomic transactions.

1. *Explicit mapping of requirements to technical problems facilitates the identification and leveraging of the necessary solution domains.*

After our requirements analysis and technical problem analysis processes as defined in sections 4.1 and 4.2 respectively, it appeared that the given client requirements did not fully describe the right detail of the desired problem. The basic requirement was to provide adaptable transactions protocols that were derived from the various expected needs of different dealers in different countries. From the initial requirement specification, however, it followed that the adaptability requirement of transaction protocols was interpreted only in a limited sense and referred to the adaptation of a restricted number of concurrency control protocols. During the problem analysis phase we generalized this requirement to the adaptation

of various transaction protocols including transaction management, concurrency control, recovery and data management techniques. After interactions with the client and a study of the car dealer distribution system it appeared that many transaction protocols were relevant although they had not been explicitly mentioned in the requirement specification. We observed that the technical problem identification is an iterative process between the technical problem analysis and solution domain analysis processes.

On the one hand, we have directed and scoped our solution domain analysis using the identified technical problems. Since every (sub-)problem corresponds only to a restricted set of solution domain we did not need to consider the whole solution domain space at once. For example, for the concept *DataManager* we did not need to consider version management and replication management because this was deliberately excluded from the scope of the project. For the concept *Scheduler* we ruled out the solution domain that dealt with semantic concurrency control techniques. The identified technical problems provided us helpful and necessary indications on where to search or not to search for the solution domain.

On the other hand, the technical problems could be better defined after the solution domains were better understood. For example, only after a solution domain analysis on concurrency control, as described in section 4.3, we were better able to accurately define the sub-problems related with the concept *Scheduler*. This observation may imply that for the problem analysis phase one may require a domain engineer who is an expert on the corresponding domain and knows the different technical problems that are related to the domain. In our example project typically a transaction domain expert at the early phase of problem analysis would be helpful.

2. *Solution domain provides stable architectural abstractions*

Synbad provides an explicit solution domain analysis process for identifying the right abstractions. After the analysis of the solution domain on transaction theory it appeared that this is rather stable and does not change abruptly but only shows a gradual specialization of the transaction concepts. Because the solution domain is stable it provides a reliable source for providing stable architectural abstractions. In the solution domain analysis process as described in section 4.3 we have illustrated how we could derive stable concepts for the design of the atomic transaction architecture. We were able to derive both the overall architecture and refine the architectural concepts to the required level of detail.

The requirement of stable solution domains in Synbad implies that a given problem can only be solved to the extent that it has been explored in the solution domain. If it appears that the solution domain is not well-established the software engineer may decide to terminate the synthesis

process, reformulate the technical problem or initiate a research on the solution domain. The latter decision shows that the synthesis process may provide important input for the scientific research because it may indicate the issues that need to be resolved in the corresponding solution domains.

3. *Solution domains provide rich semantics for realization and verification of the architecture.*

Solution domains not only provide stable abstractions but in addition these abstractions have rich semantics which is important for the realization and verification of the software architecture. As described in section 4.5 and in [35] on architecture specification, we could derive rich semantics for the architectural abstractions directly from the solution domain knowledge of atomic transactions. We have illustrated this process for various components in the atomic transaction architecture.

The solution domain is not only useful for deriving architectural abstractions, but in addition it is also a reliable source for validating the correctness of the developed architecture. We were able to identify many publications that explicitly deal with correctness proofs of various transaction protocols. We validated the architectural components and their semantics by utilizing these knowledge sources [35].

4. *Adaptability of an architecture can be determined by an explicit alternative space analysis of the solution domain.*

In Synbad, alternative space analysis is an explicit process. Thereby, for each concept the set of alternatives is described and constraints are defined among these alternatives. This together results in a depiction of the set of possible alternative designs, that is, alternatives design space, that may be derived from the given software architecture. As described in section 4.4 we have, for instance, defined the alternatives for the concepts *Scheduler* and *RecoveryManager*. From the solution domain analysis we have extracted the constraints within each of these concepts and constraints that apply among alternatives of these concepts [35]. We had two problems in the alternative space analysis process for the example project. First, although we had derived the conceptual architectures from the solution domain itself, during the alternative design process it followed that not all the alternatives were explicitly described in the literature. For example, for the concept *Scheduler* we could identify only around 10-15 scheduler types that were described in the literature. The other alternatives are primarily seen as variations of these basic scheduler types. In our approach we could depict every single alternative explicitly. The second problem that we encountered was that the constraints within and among the alternatives of the concepts are generally not explicitly stated in the literature and identifying these constraints is very

time-consuming. Defining constraints of solution domain concepts requires an improved understanding of these concepts. The existence of an explicit description of these constraints may indicate the maturity level of the corresponding solution domain. It appears that the transaction literature has many well-established concepts and we could also identify some publications that explicitly dealt with the constraints among the concepts, however, this is not the case for all the concepts.

ACKNOWLEDGEMENTS

This research has been supported and funded by various organizations including Siemens-Nixdorf Software Center, the Dutch Ministry of Economical affairs under the SENTER program, the Dutch Organization for Scientific Research (NWO, 'Inconsistency management in the requirements analysis phase' project), the AMIDST project, and by the IST Project 1999-14191 EASYCOMP.

6. REFERENCES

1. F. Ahsmann F and L. Bergmans. *I-NEDIS: New European Dealer System, Project plan I-NEDIS*, 1995.
2. M. Akşit. *Course Notes: Designing Software Architectures*. Post-Academic Organization, 2000.
3. M. Akşit, B. Tekinerdoğan, F. Marcelloni & L. Bergmans. *Deriving Object-Oriented Frameworks from Domain Knowledge*. in: M. Fayad, D. Schmidt & R. Johnson (eds.), Building Application Frameworks: Object-Oriented Foundations of Framework Design, Wiley, 1999.
4. M. Akşit, B. Tekinerdoğan and L. Bergmans. *Achieving adaptability through separation and composition of concerns*, in Max Muhlhauser (ed), Special issues in Object-Oriented Programming, Workshop Reader of the 10th European Conference on Object-Oriented Programming, ECOOP '96, Linz, Austria, July, 1996.
5. E. Arend van der. *Design of an Architecture for a Quality Management Push Framework*. MSc thesis, Dept. of Computer Science, University of Twente, 1999.
6. G. Arrango. *Domain Analysis Methods*. In Software Reusability, Schäfer, R. Prieto-Díaz, and M. Matsumoto (Eds.), Ellis Horwood, New York, New York, pp. 17-49, 1994.
7. N.S. Barghouti and G.E. Kaiser. *Concurrency Control in Advanced Database Applications*, ACM Computing Surveys, Vol. 23, No. 3, September, 1991.
8. L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*, Addison-Wesley 1998.
9. A. Bernstein and N. Goodman. *Concurrency Control in Distributed Database Systems*, ACM Transactions on Database Systems, 8(4): 484-502, 1983.
10. P.A. Bernstein and E. Newcomer. *Principles of Transaction Processing*, Morgan Kaufman Publishers, 1997.

11. P.A. Bernstein, V. Hadzilacos and N. Goodman. *Concurrency Control & Recovery in Database Systems*, Addison Wesley, 1987.
12. B.K. Bhargava (ed.). *Concurrency Control and Reliability in distributed Systems*, Van Nostrand Reinhold, 1987.
13. G. Booch, I. Jacobson and J. Rumbaugh. *The Unified Modeling Language User Guide*, Addison-Wesley, 1999.
14. P. Bourque, R. Dupuis, A. Abran, J.W. Moore and L. Tripp. *The Guide to the Software Engineering Body of Knowledge*, Vol. 16, No. 6, pp. 35-45, November/December, 1999.
15. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns*, John Wiley & Sons, 1999.
16. W. Cellary, E. Gelenbe and T. Morzy, T. *Concurrency Control in Distributed Database Systems*, North-Holland Press, 1989.
17. R.D. Coyne, M.A. Rosenman, A.D. Radford, M. Balachandran and J.S. Gero. *Knowledge-Based Design Systems*, Addison-Wesley, 1990.
18. C.J. Date. *An Introduction to Database Systems*, Vol. 3, Addison Wesley, 1990.
19. D. Diaper (ed.). *Knowledge Elicitation*, Ellis Horwood, Chichester, 1989.
20. A.K. Elmagarmid (ed.). *Database Transaction Models for AdvancedApplications Transaction Management in Database Systems*, Morgan Kaufmann Publishers, 1992.
21. M. Firlej and D. Hellens. *Knowledge elicitation: a practical handbook*, New York, Prentice Hall, 1991.
22. H. Foerster Von. *Cybernetics of Cybernetics*, in: Klaus Krippendorff (ed.), Communication and Control in Society, New York: Gordon and Breach, 1979.
23. D.D. Gajski, N.D. Dutt, A. Wu, and S. Lin. *High-level synthesis: introduction to chip and system design*, Boston : Kluwer Academic Publishers, 1992.
24. R.L. Glass and I. Vessey. *Contemporary Application-Domain Taxonomies*, IEEE Software, Vol. 12, No. 4, July 1995.
25. A.J. Gonzalez and D.D. Dankel. *The Engineering of Knowledge-Based Systems*, Prentice Hall, Englewood Cliffs, NJ, 1993.
26. J. Gray and A. Reuter. *Transaction processing: concepts and techniques*, San Mateo, Morgan Kaufmann Publishers 1993.
27. V. Hadzilacos. *A theory of reliability in Database Systems*, Journal of the ACM, 35(1): 121-145, January 1988.
28. T. Haerder and A. Reuter. *Principles of Transaction-Oriented Database Recovery*. ACM Computing Surveys, Vol. 15. No. 4. pp. 287-317, 1983.
29. R.W. Howard. *Concepts and Schemata: An Introduction*, Cassel Education, 1987.
30. I. Jacobson, G. Booch and J. Rumbaugh. *The Unified Software Development Process*, Addison-Wesley, 1999.
31. S. Jajodia and L. Kerschberg. *Advanced Transaction Models and Architectures*, Boston: Kluwer Academic Publishers, 1997.
32. P.B. Kruchten. *The 4+1 View Model of Architecture*. IEEE Software, Vol 12, No 6, pp. 42-50, November 1995.
33. G. Lakoff. *Women, Fire, and Dangerous Things: What Categories Reveal about the Mind*, The University of Chicago Press, 1987.
34. K. J. Lieberherr, *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*, PWS Publishing Company, Boston, 1996.
35. P. Loucopoulos and V. Karakostas. *System requirements engineering*, London [etc.], McGraw-Hill, 1995.
36. N. Lynch, M. Merrit, W. Weihl and A. Fekete. *Atomic Transactions*. Morgan Kaufmann Publishers, 1994.

37. M.L. Maher. *Process Models for Design Synthesis*, AI-Magazine, pp. 49-58, Winter 1990.
38. O. Maimon and D. Braha. *On the Complexity of the Design Synthesis Problem*, IEEE Transactions on Systems, Man, And Cybernetics-Part A: Systems and Humans, Vol. 26, No. 1, January 1996.
39. M. Meyer and J. Booker. *Eliciting and Analyzing Expert Judgment: A practical Guide*, Volume 5 of Knowledge-Based Systems, London: Academic Press, 1991.
40. A. Newell and. H.A. Simon. *Human Problem Solving*, Prentice-Hall, Englewood Clifss, NJ, 1976.
41. C.H. Papadimitriou. *The theory of Database Concurrency Control*. Computer Science Press, 1986.
42. J. Parsons and Y. Wand. *Choosing Classes in Conceptual Modeling*, Communications of the ACM, Vol 40. No. 6., pp. 63-69, 1997
43. D. Partridge and K.M. Hussain. *Knowledge-Based Information Systems*, McGraw-Hill, 1995.
44. G. Polya. *How to Solve It: a New Aspect of Mathematical Method*, New York, Doubleday, 1957.
45. R. Prieto-Diaz and G. Arrango (Eds.). *Domain Analysis and Software Systems Modeling*. IEEE Computer Society Press, Los Alamitos, California, 1991.
46. Y. Reich and S.J. Fenves. *The formation and use of abstract concepts in design*, in: Concept Formation: Knowledge and Experience in Unsupervised Learning, D.H.J. Fisher, M.J. Pazzani, & P. Langley (eds.), Los Altos, CA, pp. 323--353, Morgan Kaufmann, 1991.
47. E.O. Roxin. *Control theory and its applications*. Amsterdam, Gordon and Breach Science Publishers, 1997.
48. R. Rubin. *Foundations of library and information science*. New York, Neal-Schuman, 1998.
49. M. Shaw and D. Garlan. *Software Architectures: Perspectives on an Emerging Discipline*,. Englewood Cliffs, NJ: Prentice-Hall, 1996.
50. M. Shaw. *Moving from Qualities to Architectures: Architectural Styles*, in: L. Bass, P. Clements, & R. Kazman (eds.), Software Architecture in Practice, Addison-Wesley, 1998.
51. I. Sommerville and P. Sawyer. *Requirements engineering: a good practice guide*, Chichester, Wiley, 1997.
52. N.A. Stillings, S.E. Weisler, C.H. Chase, M.H. Feinstein, J.L. Garfield and E.L. Rissland. *Cognitive Science: An Introduction.* Second Edition, The MIT Press, Cambridge, Massachusetts, 1995.
53. B. Tekinerdoğan. *Synthesis-Based Software Architecture Design*, PhD Thesis, Dept. Of Computer Science, University of Twente, March 23, 2000.
54. B. Tekinerdoğan. *Overall Requirements Analysis for INEDIS,* Siemens-Nixdorf/University of Twente, INEDIS project, 1995.
55. B. Tekinerdoğan. *Requirements for Transaction Processing in INEDIS*, Siemens-Nixdorf/University of Twente, INEDIS project, 1995.
56. B. Tekinerdoğan. *Reliability problems and issues in a distributed car dealer information system*, INEDIS project, 1996.
57. B. Tekinerdoğan and M. Akşit. *Adaptability in object-oriented software development, Workshop report*, in M. Muhlhauser (ed), Special issues in Object-Oriented Programming, Dpunkt, Heidelberg, 1997.

58. B. Tekinerdoğan and M. Akşit. *Deriving design aspects from conceptual models*. In: Demeyer, S., & Bosch, J. (eds.), Object-Oriented Technology, ECOOP '98 Workshop Reader, LNCS 1543, Springer-Verlag, pp. 410-414, 1999.
59. R.H. Thayer, M. Dorfman and S.C. Bailin. *Software requirements engineering*, Los Alamitos, IEEE Computer Society Press, 1997.
60. W. Tracz and L. Coglianese. *DSSA Engineering Process Guidelines*. Technical Report, ADAGE-IBM-9202, IBM Federal Systems Company, December, 1992.
61. I.L. Traiger, J. Gray, C.A. Caltiere and B.G. Lindsay. *Transactions and Consistency in Distributed Database Systems*, ACM Transactions on Database Systems, Vol. 7, No. 3, pp 323-342, September, 1982,
62. S.A. Umplebey. *The Science of Cybernetics and the Cybernetics of Science, Cybernetics and Systems*, Vol. 21, No. 1, 1990, pp. 109-121, 1990.
63. C. Vuijst. *Design of an Object-Oriented Framework for Image Algebra*. MSc thesis, Dept. of Computer Science, University of Twente, 1994.
64. J.B. Warmer and A.G. Kleppe. The Object Constraint Language : Precise Modeling With Uml, Addison-Wesley, 1999.
65. W. Weihl. The impact of recovery on concurrency control. Proceedings of the eigth ACM SIGACT-SIGMOD-SIGART symposium on Principles of Database Systems March 29 - 31, Philadelphia, PA USA, 1989.
66. B.J. Wielinga, T. Schreiber and J.A. Breuker. KADS: a modeling approach to knowledge engineering, Academic Press, 1992.
67. R. Willems. *Ontwikkelen van verzekeringsproducten*, dutch, translation: Development of Insurance Products, MSc thesis, Dept. of Computer Science, University of Twente, 1999.
68. Z. Wu, R.J. Stroud, K. Moody and J. Bacon. *The design and implementation of a distributed transaction system based on* atomic *data types*, Distributed Syst, Engineering, 2, pp. 50-64, 1995.

Chapter 6

LOOSELY COUPLED COMPONENTS

Patrick Th. Eugster*, Rachid Guerraoui* and Joe Sventek**

* *Department of Communication Systems, Swiss Federal Institute of Technology, Lausanne.*
Communication Solutions Department, Agilent Laboratories Scotland, Edinburgh.
email: {Patrick.Eugster, Rachid.Guerraoui} @epfl.ch
** *Communication Solutions Department, Agilent Laboratories Scotland, Edinburgh.*
email: sventek@labs.agilent.com

Keywords: Distributed components, abstraction, asynchronous interaction, message queuing, collections publish/subscribe

Abstract: Collections are widely used as a basic programming abstraction to store, retrieve and manipulate objects. There are different known types of collections (e.g., sets, bags, queues), offering various semantics for different application purposes. A collection can offer a distributed flavor, that is, it can be accessible from various nodes of a network. The elements of such a collection are thus shared between the different nodes, and a distributed collection can be viewed as a means of exchanging information between components, in a way similar to a shared memory. This chapter presents Distributed Asynchronous Collections (DACs). Roughly spoken, a DAC is capable of calling back an interested party in order to notify for instance the insertion or removal of elements. By viewing the elements of our Distributed Asynchronous Collections (DACs) as events, these collections can be seen as programming abstractions for asynchronous distributed interaction, enabling the loose coupling of components. In that sense, the DACs we present in this chapter marry the two worlds of object-orientation and event-based, so-called "message-oriented", middleware. DACs are general enough to capture the commonalities of various message-oriented interaction styles, like message queuing and publish/subscribe interaction, and flexible enough to allow the exploitation of the differences between these styles.

1. INTRODUCTION

This chapter presents *Distributed Asynchronous Collections* (DACs), simple object-oriented abstractions for expressing the many diverging message-oriented interaction ("messaging") styles.

Motivation. With the emergence of wide area networks, the importance of flexible, well-structured and also efficient communication mechanisms is increasing. Basing a complex interaction between multiple nodes on individual *point-to-point* communication models is a burden for the application developer and leads to rather static and limited applications. In mobile communications furthermore, it may not be simple for an application to spot the exact location of a component at any moment. Also the number of components interested in certain information may vary throughout the entire lifetime of the system. All these constraints visualize the demand for more flexible communication models, reflecting the dynamic nature of the applications. The *publish/subscribe* and message queuing interaction styles satisfy those requirements.

Messaging dialects. There are different established message-oriented interaction models, each one presenting its respective advantages but also shortcomings. The publish/subscribe interaction style involves a decoupling in *time* and in *space* of information producers and consumers[1]: producers publish information on a software bus while consumers subscribe to that information bus [1] and are asynchronously notified of new information. The decoupling nature of publish/subscribe is not only important for enterprise computing products, but also for many emerging e-commerce and telecommunication applications.

The classical *topic-based* or *subject-based* publish/subscribe style involves a static classification of the messages by introducing group-like notions [2], and is now incorporated by most industrial strength solutions, e.g., [3, 4, 5, 6]. It is based on a *push model*, and messages are consumed by several subscribers, i.e., message are dispatched on an *all-of-n* (*one-for-all*) base to the consumers.

Message queuing [7, 8, 9, 10] is an approach which traditionally combines *pull-style* mechanisms on the consumer side with *one-of-n* semantics: messages are pushed into a queue and several consumers concurrently pop messages from that queue. A message in general represents

1 Time decoupling: the interacting parties do not need to be up at the same time. Space decoupling: the interacting parties do not need to know each other.

a request and is processed by a single consumer. This interaction scheme is thus often used for public services, like health care, banking and finance.

As noticed in [11] in fact, some applications need pull- *or* push-style while others require *both*. The same way, applications may require all-of-n *or* one-of-n semantics, *both* with push- or pull-style interaction. Instead of bringing all these variants to a common denominator, much emphasis is usually put on their differences. The DACs we present in this chapter are simple programming abstractions, which capture the different message-oriented interaction styles, without blurring their respective advantages.

Messaging abstractions. A DAC differs from a conventional collection by its distributed nature and the way objects interact with it: besides representing a collection of objects (*set*, *bag*, *queue*, etc.), a DAC can be viewed as a messaging system of its own. In fact, when querying a DAC for objects fulfilling certain conditions, the client expresses its interest in such objects. In other words, the invocation of an operation on a DAC enables the expression of *future notifications* and can be viewed as a subscription. According to the terminology adopted in the *observer design pattern* [12], the DAC is the *subject* and its client is the *observer*.

Contributions. In short, within all messaging models none is clearly better than the others for all application purposes. We illustrate in this chapter how DACs can be used to express the different message-oriented interaction styles, through the examples of topic-based publish/subscribe and message queuing. We visualize furthermore how DACs enable the unification of these messaging flavors in a single framework. Our DAC framework can be seen as an extension of a conventional collection framework. It enables to mix push and pull models, all-of-n and one-of-n semantics along with different *qualities of service* (QoS). Our Java implementation of the DAC framework is fully integrated with the standard Java collection framework, and can be seen as an extension of latter one.

Roadmap. The remainder of this chapter is organized as follows. Section 2 recalls the various interaction styles in distributed computing and motivates the need for weakly coupled communication models. Section 3 gives an overview of our DAC abstraction. Section 4 gives the basic DAC API for topic-based publish/subscribe, whereas Section 5 shows the API's enabling the expression of message queuing. Section 6 gives a simple programming example using DACs for topic-based publish/subscribe. In Section 7 we discuss some performance issues of our implementation, and Section 8 discusses several design issues. Section 9 contrasts our efforts with

related work. Finally Section 10 summarizes our work and concludes the chapter.

2. MESSAGE-ORIENTED INTERACTION STYLES: COMMONALITIES AND VARIATIONS

Before describing our DAC abstraction, we first overview the basics of message-oriented interaction styles, by contrasting publish/subscribe communication and message queuing with more traditional interaction schemes. In addition, we compare the different flavors of messaging in more detail. We point out the fact that each of the different variants has proven certain advantages over others, which motivates the usefulness of unifying them inside a single framework.

2.1 Publish/subscribe in Perspective

The publish/subscribe paradigm is a loose communication scheme for modeling the interaction between components in distributed systems. Unlike the classic *request/reply* model or *shared memory* communication, publish/subscribe provides *time decoupling* (i.e., the interacting parties do not need to be up at the same time) of message producers and consumers. Figure 1 shows a comparison of the most common communication schemes: *message passing* (*singleton send*) may also offer an asynchronous interaction scheme, but lacks *space decoupling* (i.e., the interacting parties need to know each other), just like the request/reply communication style. Indeed with message passing, the information producer must have a means of locating the information consumer to which the information will be sent, whereas with the request/reply interaction model the consumer requires a reference to the information producer in order to issue a request to it[2]. Publish/subscribe combines *time* as well as *space* decoupling, since the information providers and consumers remain anonymous to each other. This outlines the general applicability of this communication model and makes it appealing[3]. Like communication based on shared memory, publish/subscribe moreover allows

[2] In most middleware solutions based on remote method invocations, references are by default transient, i.e., an object which is destroyed and recreated (or migrated) restarts with a new identity. Migration of objects requires specific support [35].

[3] It is possible to build closer coupled communication models on top of loose ones and vice versa, as proposed by [36] for instance. The resulting performance in the second case however is generally poor.

to address several destinations *(arity of n)*. Basically the publish/subscribe terminology defines two players:

- *Subscriber*: A party which is interested in certain information (events, messages) subscribes to that information, signaling that it wishes to receive all pieces of information (event notifications, messages) manifesting the specified characteristics. A subscriber party can be seen as a consumer. *Leasing* is a special form of subscribing, in which the duration of the subscription is limited by a time-out.
- *Publisher*: A party that produces information (events, messages) becomes a *publisher*.

In most applications however, participating entities incorporate both publishers and subscribers, which allows a very flexible interaction. This is one of the main differences to pure *push-based* systems [13], where participants act either as producers or as consumers and producers are supposed to be several orders of magnitude higher in number than consumers.

	Time	Space	Arity
Request/Reply	Coupled	Coupled	1
Singleton Send	Decoupled	Coupled	1
Shared Memory	Coupled	Decoupled	n
Message Queuing	Decoupled[(a)]	Decoupled	n[(b)]
Publish/Subscribe	Decoupled	Decoupled	n

Figure 1: Different communication models

2.2 Topic-based Publish/subscribe

When subscribing, a party expresses its interests in receiving certain messages. Rarely a subscriber is eager to receive all produced messages. Dividing the message space provides a means of confining the subscribers requirements. The classic publish/subscribe interaction model is based on the notion of *topics* or *subjects*, which basically resemble groups [2]. Subscribing to a topic *T* can be viewed as becoming member of a group *T*. The topic abstraction however differs from the group abstraction by its more dynamic nature. While groups are usually disjoint sets of members (e.g., group communication for replication [14]), topics typically overlap, i.e., a participant subscribes to more than just one topic. In order to classify the topics more easily, it is of great use to furthermore arrange them in a hierarchy, e.g., [4, 6]. In this model, a topic can be a derived or more

specialized topic of another one, and is therefore called *subtopic*. The use of wildcards offers a convenient way of expressing *cross-topic* requests.

Figure 2 shows an example of topic-based subscribing. Subscriber S_1 subscribes to topic */Chat* and claims its interest in all subtopics. Hence S_1 does not only receive message m_2 but also message m_1 published for topic */Chat/Insomnia*. In contrast, S_2 only subscribes to */Chat/Insomnia* and thus does not receive message m_2, which belongs to the supertopic */Chat*.

2.3 Message Queuing

In the publish/subscribe model, the action of subscribing describes a sort of registration procedure for an interested party. Interests in events can also be expressed through a more direct interaction. *Message queuing* is a well-known alternative to publish/subscribe interaction, where consumers actively pull messages from a shared queue. Such queues recall much the shared memory model and their derivatives, e.g., *tuple space* [15]. In message queuing, as the name reveals, messages are delivered to pullers in a FIFO order, and every message is usually processed by a single consumer. In contrast to the *all-of-n* (*one-for-all*) semantics found with the publish/subscribe interaction style, this is frequently called *one-of-n* [4][4]. This one-of-n scheme is often used when requests have to be processed and several servers are eligible to process those requests, but the same request should not be processed more than once.

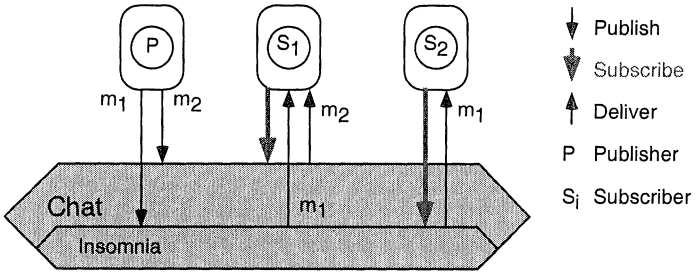


Figure 2: Topic-based subscribing

The pull-style interaction of the consumers with a given queue can take place in two ways:

[4] By using the formalism of [37], one could say that every Nth occurrence of an event is notified to a consumer, with N being the total number of consumers, and no event being delivered to more than one consumer.

– *Polling*: A consumer can *poll* for new notifications. This task may waste resources and is not well adapted to asynchronous systems. In fact, polling based solutions tend to be very expensive and scale poorly: polling too often can be inefficient and polling too slowly may result in delayed responses to critical situations [5].
– *Blocking pull*: Another yet more synchronous pull-type interaction is given by *blocking* pull-style interaction. In this scenario, a consumer which tries to pull information is blocked until a new notification is available. Just like the request/reply model however, this variant introduces time coupling.

The way we have reported message-queuing in Figure 1 requires two clarifications. *(a)*, the classical pull-style interaction of consumers with the queue introduces a time coupling. This is however not a necessity, and as we will see later, nothing prevents from using a push model in this context to enforce time decoupling. *(b)*, it can involve several consumers (n), with the above mentioned restriction that a message is processed by a single consumer.

Figure 3 shows an example. Suppose that P inserts first m_1 and then m_2. When C_1 or C_2 pull the messages inserted by P, they are synchronized with the queue. In this case, C_1's request is served first, i.e., m_1 is returned to it. Synchronization even occurs if the consumers are immediately released in the absence of new message. On the other hand, producers are not coupled in time, since they asynchronously notify events to the queue.

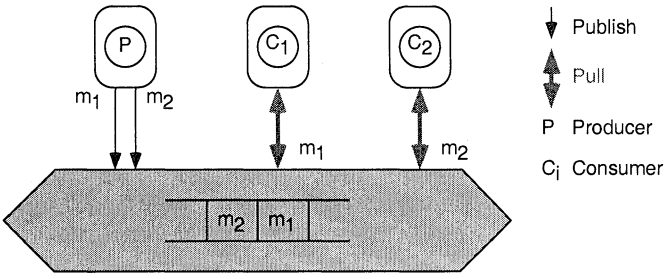


Figure 3: Message queuing

2.4 Mixing Push and Pull

The essential difference between message queuing and publish/subscribe is that in the former case an event is notified to *one* consumer only, while with the latter style an event is notified to *each* consumer. The choice of pull- or push-style interaction on the consumer side however is orthogonal to

the number of receivers. The *CORBA Event Service* [16] is an interesting approach to publish/subscribe in the sense that it enables the mixture of pull- and push-style consumers *and* producers.

In that sense, a general framework should relax restrictions of classical message-oriented styles. With all-of-n semantics for instance, nothing prevents from promoting pull-style interaction on the consumer side, and even enabling concurrent push-style consumers. In the case of one-of-n semantics, messages can also be pushed from the queue towards the consumers. As we will show later, combining with concurrent pulling consumers in that case is however more difficult.

The overall time coupling introduced by pulled producers however has a higher impact on the system than pull-style interaction on the consumer side, and should thus be avoided. Indeed, with pulling consumers, the synchronization can be restrained more locally, but the synchronization introduced by pulling producers propagates to the entire system. In the following, we will thus always consider producers acting in push mode, and ignore *pullsuppliers* (in the terminology of [16]). The question of push- or pull-style interaction will only be raised with respect to the consumer side.

2.5 Delivery Semantics and Reliability Issues

In distributed systems, and in particular when considering communication models and protocols, precise specification of the semantics of a delivery is a crucial issue. Delivery guarantees are often limited by the behavior of deeper communication layers, down to the properties of the network itself, limiting the choice of feasible semantics. On the other hand, different applications also may demand for different semantics. While sometimes a high throughput is pre-eminent and a low reliability degree is tolerable, some applications prioritize reliability to throughput. For this reason, most common systems provide different *qualities of service* (QoS), in order to meet the demands of a variety of application purposes [4, 6][5]. The guarantees offered by existing systems can be roughly divided into two groups.

– *Unreliable delivery*: Protocols for unreliable delivery give few guarantees. These semantics are often used for applications where the throughput is of primary importance, but the loss of certain messages is not fatal for the application.

[5] [4] adopts the notion of delivery service.

– *Reliable delivery*: Reliable delivery means that a message will be delivered to every subscriber despite certain failures. Usually the failure or the absence of the subscriber itself is not considered, i.e., if the subscriber has failed, the message might not be delivered to it and the reliability property is not considered violated. When using persistent storage to buffer such messages until the subscriber is back on line, a stronger guarantee is given. This is sometimes referred to as *certified delivery*, [4] or *guaranteed delivery* [9].

3. DISTRIBUTED ASYNCHRONOUS COLLECTIONS: OVERVIEW

This section gives an overview of our approach to messaging, by first introducing *Distributed Asynchronous Collections* as key abstractions. We show the relationship between these abstractions and message-oriented communication models. This allows discussing in the following sections how these abstractions allowed us to build several different messaging styles inside a unified framework.

3.1 DACS as Object Containers

Just like any collection, a DAC is an abstraction of a container object that represents a group of objects. It can be seen as a means to store, retrieve and manipulate objects that form a natural group, like a mail folder or a file directory. Unlike a conventional collection, a DAC is a distributed collection whose operations might be invoked from various nodes of a network. In contrast to remotely accessible collections like the ones described in [17] however, DACs are asynchronous and essentially distributed: DACs are not centralized on a single node, in order to guarantee their availability despite certain failures[6]. Instead, they can be viewed as replicated or fragmented. Participating processes act with a DAC through a local proxy. The proxy is viewed as a local collection and hides the distribution of the DAC, as shown in Figure 4. The proxy acts in a way similar to a cache, and the type of the DAC depends on the consistency between the respective local proxies. As a consequence of distribution, a globally unique name is assigned to every

[6] The distributed collections presented in [17] are centralized collections that can be remotely accessed through RMI. Similarly, the OMG has defined a collection framework as part of the CORBA services specification [16]. The specification aims at a centralized model, but distributed implementations are not ruled out.

DAC. This provides a uniform manner for participants to designate the DAC(s) they wish to "connect" to.

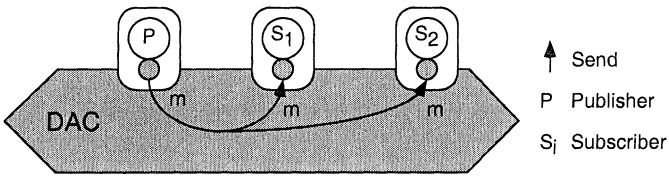


Figure 4: DAC distribution

3.2 The Asynchronous Flavor of DACS

Our notion of Distributed Asynchronous Collection represents more than just a distributed collection. In general, a synchronous invocation of a distant object can involve a considerable latency, hardly comparable with that of a local one. In contrast, asynchronous interaction is supported with our collections. By calling an operation of a DAC, one can express an interest in *future notifications*. When querying a DAC for objects of a certain kind, the party interacting with the DAC expresses its interest in such objects. In the case of pull-style interaction, the DAC replies to the querying party. The DAC can also register the interest and defer the reply (replies): the interested party is asynchronously notified when an object matching the query is eventually "pushed" into the DAC.

There is a strong resemblance between such a push model and the notion of *future* [18] (*future type message passing* [19]), that describes a communication model in which a client queries an *asynchronous object* for information by issuing a request to it. Instead of blocking however, the client can pursue its processing. As soon as the reply has been computed, the object acting as server notifies the client. Latter one may query the result (*lazy synchronization* or *wait-by-necessity* [20]), or ignore it. Figure 5 compares the two paradigms. When programming with DACs, the subscriber can be viewed as the client. The DAC incarnates a server role in this scenario, since the publishers, which are the effective information suppliers, remain anonymous.

By calling an operation on the DAC, the caller requests certain information. The main difference with futures lies in the number of times that information is supplied to the client. Within the notion of future, only a single reply is passed to the client,[7] whereas with DACs, every time an

[7] ABCL/1 represents an exception, in the sense that several replies may be returned [19].

information which is interesting for the registered party is created, it will be sent to it.

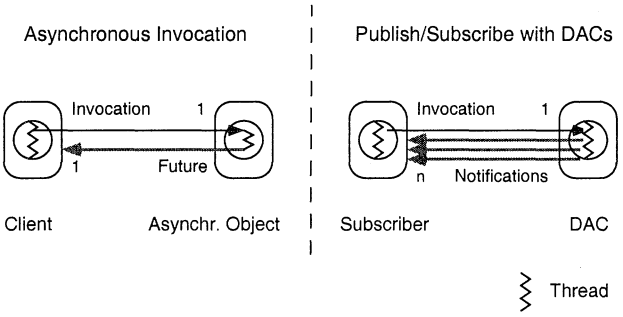


Figure 5: DACs vs. future

4. TOPIC-BASED PUBLISH/SUBSCRIBE WITH DACS

Traditionally, topic-based publish/subscribe enables a strong decoupling of participants by involving a strongly asynchronous interaction. Furthermore it provides for platform interoperability, by relying only on topic names (strings) as references for distributed participants to meet. In this section we use topic-based publish/subscribe as a first example of a messaging style that can be expressed with DACs.

4.1 Collections and Topics

Expressing ones interest in objects of a certain kind can be viewed as subscribing to objects of that kind. When considering the classical topic-based approach to publish/subscribe, a DAC can be pictured as an extension of a conventional collection and also as a representation for a topic. It is always possible to insert a new element into a DAC. In the sense of publish/subscribe, inserting an object into a DAC also means to publish that object for the topic represented by the DAC. Every DAC can thus be viewed as a publish/subscribe engine of its own.

In our system, each topic is denoted by a name, like *Chat*. Topics can have specializations, or *subtopics*, and connecting to a topic requires the name in a URL-type format. Typically, */Chat/Insomnia* is a reference to the topic called *Insomnia* which is a subtopic of *Chat*. The root of the hierarchy is represented by an abstract topic (denoted by /). Top-level topics, which are no specializations of already existing ones, are subtopics of the abstract root

topic only. The all-of-n semantics of topic-based publish/subscribe enables to easily trigger subscriptions to subtopics as well.

4.2 DAC Interfaces: Overview

We present the main interfaces related to topic-based publish/subscribe of our DAC realization in Java. In the context of this chapter, we will limit ourselves to describing basic functionalities which are common to subinterfaces, in order to show their similarity to operations on conventional centralized collections.

Messages. Existing publish/subscribe frameworks introduce specialized message types, e.g., [21]. Our approach frees the application programmer from the burden of marshalling and unmarshalling data into and from dedicated messages. In our context, a message can be basically of any kind of object. In Java, this is expressed by allowing any object of class `java.lang.Object` to be passed as a message[8].

Callbacks. In the case of push-style interaction between a DAC and a consumer, the DAC will call back the consumer in order to asynchronously notify it of events. Consumers acting in a push mode must for that purpose provide an object which implements the `Notifiable` callback interface given in Figure 6. It extends the `ExceptionHandler` interface, which any participant (pull or push) must provide in order to learn about and react to exceptions. Furthermore, this allows the DAC to know the number of effectively "connected" participants.

```
public interface ExceptionHandler {
        public void handleException  (Exception ex,
                                      String DACName);
    }
    public interface Notifiable
        extends ExceptionHandler {
        public void notify  (Object m, String DACName);
    }
```

Figure 6: Callback interfaces for DACs

8 In order to be conveyable, a Java object should furthermore implement the java.io.Serializable interface [38], which contains no methods.

DAC interface hierarchy. Similarly to the Java collection framework, our DAC framework presents several interfaces which form a hierarchy. The root of the hierarchy is the `DACollection` interface (Figure 7). It inherits from the standard Java `java.util.Collectioninterface`, since a DAC is in the first place a collection. It mainly adds functionalities related to its distributed nature: a DAC is characterized by a name, with allows it to be referenced from any node.

The DAC hierarchy is roughly split in two subhierarchies, representing publish/subscribe interaction and message queuing respectively. This bipartition is given by the different interaction styles. In fact, publish/subscribe enables an easy combining of both pull- and push-style interacting consumers at the topic level. A puller will register its interest just like a subscriber, but without being called back. It will then pull notifications, and will thus in principle be notified of all produced events. This mixture is possible thanks to the all-of-n flavor of the topic-based publish/subscribe scheme we adopt. We will see in the next section that this is not straightforward in the case of message queuing.

Distributed Asynchronous Sets. The interface `DACollection` is thus used to express common functionalities of DAC types for all messaging styles offered by our framework. A DAC for publish/subscribe is of a more specific type, *Distributed Asynchronous Set* (DAS). The corresponding interface, `DASet`, is given in Figure 7.

4.3 Basic Methods

Figures 7 summarizes the main methods of the basic interfaces involved in our topic-based publish/subscribe scheme. Methods inherited from Java collections are not denatured but adapted, and we connote them as *synchronous* in contrast to the *asynchronous* methods added mainly to express consumer-side push-style interaction.

Synchronous methods. These synchronous methods are adapted from centralized collections:

- `contains(Object)`: A DAC is first of all a representation of a collection of elements. This standard method allows querying the collection for the presence of an object.
- `add(Object)`: This method allows to add an object to the collection. The corresponding meaning for a DAC is straightforward: it allows publishing a message for the topic represented by that collection. An

asynchronous variant of this method (registering a callback object) would consist in advertising the eventual production of notifications and would lead to pulling producers.

– `get(...)`: Similarly to a centralized collection, calling this method on a DASet allows to retrieve objects. This implements the pull model. The variant with a timeout argument represents a blocking pull, while the argumentless counterpart expresses polling.

```
public interface DACollection
    extends java.util.Collection
{   /* from original Collection interface */
    public boolean contains(Object message);
    public boolean add(Object message);
    ...
    /* distribution obliges */
    public String getName();
    public boolean hasName(String name);
    ...
}
public interface DASet
    extends DACollection, java.util.Set
{   /* pull-style: register */
    public boolean contains(ExceptionHandler h);
    public boolean containsAll(ExceptionHandler h);
    /* pull-style interaction */
    public Object get(ExceptionHandler h);
    public Object get(ExceptionHandler h,
                      long timeout);
    ...
    /* push-style: subscribe */
    public boolean contains(Notifiable n);
    public boolean containsAll(Notifiable n);
    ...
    /* unregister/unsubscribe */
    public void clear(ExceptionHandler h);
    ...
}
```

Figure 7: Interfaces `DACollection` and `DASet` (Excerpts)

Asynchronous methods. The following *asynchronous* methods in the `DASet` interface reflect the decoupling nature of topic-based publish/subscribe specific to DACs.

- `contains(...)`: The goal of this method without `Object` argument is not to check if the collection already contains an object revealing certain characteristics, but is to manifest an interest in any such object, that should be eventually pushed into the collection. This method serves as a registration procedure for an interested party. It comes with two signatures: (1) an argument of `Notifiable` signals that the registering party is a subscriber which wishes to be asynchronously notified of new events. (2) In the case of a pulling client, an `ExceptionHandler` must be provided. In any case, the registration expresses all-of-n semantics.
- `containsAll(...)`: This method offers the same signatures than the previous method. The difference is that a subscription is generated for all subtopics of the topic represented by this DAS.
- `clear(...)`: While the conventional argument-less `clear()` method allows to erase all elements from the collection, this asynchronous variant expresses the action of *unsubscribing*, resp. *unregistering*.

The following section introduces more asynchronous methods related to message queuing.

5. MESSAGE QUEUING WITH DACS

Besides publish/subscribe, message queuing is another widely adopted messaging style. In this section, we present how DACs provide a natural way of expressing message queuing.

5.1 Distributed Asynchronous Queues

By viewing event notifications as objects, a DAC can be seen as an entity representing related objects or event notifications. The queuing interaction model fits intuitively into this philosophy: the abstraction used for message queuing is the queue, which is a specific collection type. *Distributed Asynchronous Queues* (DAQs) are a specific type of DACs, used to reflect message queuing. What distinguishes them from other DACs is their one-of-n flavor,which ensures that a message object pushed into a DAQ will not be consumed by more than one consumer.

Inheritance issues. Queues (centralized ones) are usually seen as a special kind of sequences, a subtype of sets, since the elements are strictly ordered. However, we use sets and sequences to model topic-based publish/subscribe, whilst queues represent message queues, involving different interaction types. While, push- and pull-style invocations can be easily mixed in the case of topic-based publish/subscribe, the one-of-n semantics associated with message queues makes a mixture difficult: non-blocking consumers would be strongly penalized, since incoming messages would be instantaneously pushed towards subscribers. Therefore the `DAQueue` interface does not inherit from `DASet` (Figure 8).

Asynchrony issues. In our framework, classical pull-style queues implement the `DABasicQueue` interface. Combining one-of-n semantics with push-style interaction on the consumer side is very interesting since it enforces asynchronous interaction. Figure 8 shows also the `DACallbackQueue` interface, which notifies consumers of new messages. By default, a round-robin policy is applied, but specialized subtypes distribute the messages according to more subtle policies, e.g., proportionally to priorities associated to the consumers.

Methods. A consumer expresses its interest in messages of a DAQ by means of the following methods:

- `push(Object)`: This method is invoked to insert a new element into the queue. It has the same effect than invoking the `add()` method on the queue.
- `remove(...)`: The effect of invoking this method is not to trigger the removal of an object already contained in the queue, but to express an interest in being notified whenever a matching object is inserted in the collection. The object that is returned to a consumer is immediately removed thereafter. In the case of a callback queue, a callback object is registered.
- `pop(...)`: In the case of a pull-style queue, the consumer must interact more directly with the queue to receive objects. This is done by pulling the queue through this method. The variant with timeout argument expresses a blocking pull interaction.

5.2 Subqueues

DACs can be used to represent message queues as well as publish/subscribe channels. Besides push-style interaction of consumers with

queues, other publish/subscribe features can be transposed to message queues as well.

```
public interface DAQueue
        extends DACollection, java.util.Set
    { .../* same effect than "add" */
        public boolean push(Object obj);
        ...
        /* unregister/unsubscribe */
        public void clear(ExceptionHandler h);
        ...
    }
    public interface DABasicQueue
        extends DAQueue
    { .../* pull-style: register */
        public boolean remove(ExceptionHandler h);
        /* pull-style: register with subqueues */
        public boolean removeAll(ExceptionHandler h);
        ...
        /* pull-style interaction */
        public Object pop(ExceptionHandler h);
        public Object pop(ExceptionHandler h, long timeout);
        ...
    }
    public interface DACallbackQueue
        extends DAQueue
     { .../* push-style: subscribe */
        public boolean remove(Notifiable n);
        /* push-style: subscribe with subqueues */
        public boolean removeAll(Notifiable n);
        ...
    }
```

Figure 8: Interfaces `DAQueue`, `DABasicQueue` and `DACallbackQueue` (Excerpts)

Arranging topics in a hierarchy enables to automatically generate subscriptions to subtopics. The same way, a subscription to a DAQ can be applied to its subordinates. DAQs, wether for push- or pull-style consumers, are arranged in a hierarchical name space. When pulling a DAQ Q and its *subqueues*, Q is checked for new values. If none is available, Q's subqueues

are successively checked. Similarly to the `containsAll()` method in the `DASet` interface, we have for that purpose added the `removeAll()` method. The names for DAQs hence follow a URL-like notation. Because of the differences between topics and queues, and between pull-style and push style queues, the different corresponding collection types involve different naming hierarchies: subscribing to a topic and its subcollections does not make much sense if these are queues used in pull-mode.

As a consequence of this separation there might very well be a topic */Chat/Insomnia* as well as a queue with the same name */Chat/Insomnia*. Conflicts between the different semantics are avoided by using separate name spaces for (1) topics, (2) pull queues and (3) push queues.

6. PROGRAMMING WITH DACS

This section describes a simple example application using the flexibility of Distributed Asynchronous Collections. It shows how to implement *chat sessions* based on simple DACs for topic-based publish/subscribe. We will concentrate on two users, Alice and Tom. They are both chat addicts, and love to chat deep into the night. Therefore they subscribe to the topic *Insomnia* which is a subtopic of *Chat* to receive all messages from like-minded chatters (see Figure 9). For the sake of simplicity, we will assume that this evening Tom is missing inspiration, and therefore takes a pure subscriber role. Alice on the other hand, is very talkative, and publishes several messages. Figure 10 shows class `ChatMsg`, which represents a possible message class for this application.

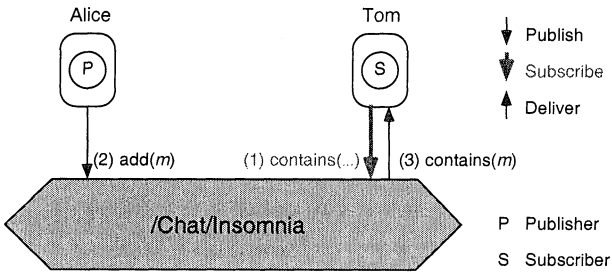


Figure 9: Chatters

```
public class ChatMsg
    implements java.io.Serializable
  { private String sender;
    private String text;
    public String getSender() { return sender; }
    public String getText() { return text; }
    public ChatMsg(String sender, String text) {
      this.sender = sender; this.text = text; }
}
```

Figure 10: Event class for chat example

6.1 Publishing for a Topic

When making use of topic-based publish/subscribe, a topic is represented by a DAS, as seen previously. In order to access a DAS from a process, a proxy must be created. This requires an argument denoting the name of the topic it bears. To the application, the action of creating a proxy is similar to creating a local collection, except that a name must be provided as argument. The DAC instance called *mychat* in Figure 11 henceforth allows us to access the topic */Chat/Insomnia*. The instantiated class `DAStrongSet` is an implementation which provides reliable delivery of messages to all subscribers. Now that a proxy has been created, it is possible to directly publish and receive messages for the topic associated to that DAS.

Creating an event notification for a topic consists in inserting a message object into the DAC by issuing a call to the `add()` method (see Section 4), from where it becomes accessible for any party.

```
DASet mychat =
  new DAStrongSet("/Chat/Insomnia");
String me = "Alice";
ChatMsg m = new ChatMsg(me, "Hi everyone");
mychat.add(m);
```

Figure 11: Publishing a message

6.2 Subscribing to a Topic

In this context, it is more favourable for Tom to be notified automatically when a new message has been published, than to waste computation time on polling activity. In order to subscribe to a topic an interested party must provide a callback object implementing the `Notifiable` interface. The

callback method comprises two arguments. The first argument represents the effective message, and the second argument represents the name of the DAC the message was published for. This provides more flexibility, since the same subscriber object can be used to receive messages related to several topics. As shown in Figure 12, Tom is only interested in a specific chat session *Insomnia*. Otherwise, Tom would have subscribed by using `containsAll()`.

7. IMPLEMENTATION ISSUES

This section discusses the realization of our first DAC implementation, including performance measurements made in the context of topic-based publish/subscribe. We draw preliminary conclusions of our prototype, which has been developed in pure Java and relies on UDP, thus increasing its portability.

```
class ChatNotifiable
    implements Notifiable
{ public void handlException(Exception ex,
                             String DACname) {
        ex.printStackTrace();
    }
    public void notify(Object m, String topic) {
        System.out.println(((ChatMsg)m).getText());
    }
}
DASet sleeplessChatters =
    new DAStrongSet("/Chat/Insomnia");
Notifiable n = new ChatNotifiable();
sleeplessChatters.contains(n);
...
```

Figure 12: Topic-based publish/subscribe with DACs

7.1 Inside DACS

The effective DAC class as it is perceived by the application only represents a small portion of the underlying code. Redundant code has been avoided by a modular design and using inheritance. Figure 13 shows the

different layers in our implementation. These layers do not necessarily correspond to Java classes, but represent protocol layers.

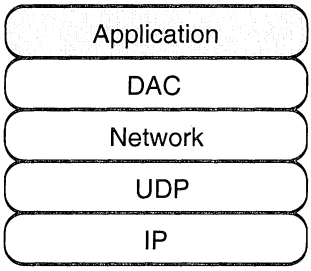


Figure 13: Layers

- *The DAC layer*: This layer is composed of the classes implementing directly the DAC interfaces. They are rather lightweight classes, which delegate general functionality to the underlying layer. Their tasks are similar to centralized container classes. They mainly take care of the local management of messages, and furthermore handle the subscriptions. The most frequent interaction model is the callback model (push-model), where subscribers do not poll for new messages but are called back upon incoming messages. In that case the DAC applies a predefined threading model, by assigning notifications to threads.
- *The Network layer*: The Network layer regroups common functionalities of all DACs, like publishing messages or forwarding subscription information. It hides any remote party involved in same topics from the DAC layer. This layer maintains a form of network topology knowledge, which basically consists of its immediate neighbors.
- *The UDP layer*: Our entire messaging architecture is finally implemented on top of UDP. UDP is a non reliable protocol, which offers us the looseness required for asynchronous interaction. Java offers classes for UDP sockets and datagrams (`java.net.DatagramPacket` and `DatagramSocket`), which are pretty close to the metal.

7.2 Performance

The performance tests of our prototype were made on HP workstations running HP-UX 10.20 and JVM 1.1.5 and 1.1.6. on a normal working day. The implementation uses a marshalling/unmarshalling procedure built from scratch and optimized for each event type (the Java serialization classes were not used,since they are usually considered rather slow). Four example message types were considered with all-of-one semantics (topic-based publish/subscribe):

- `Integer`: This corresponds to the basic Java `int` type.
- `String`:: Java type `String` with a length varying between 10 and 20.
- `DetailRecord`: This is a class containing four attributes, of which two represent dates (Java type `Date`) and two are strings (Java type `String`).
- `CallDetailRecord`: A subtype of `DetailRecord`. In addition to the attributes of the latter one, a `CallDetailRecord` furthermore contains 4 integers and two strings.

In our measurement scenario, several subscribers asynchronously receive events for a topic where a publisher produced the events. The numbers of messages considered for a single run of the experiment varied between 10 and 1000 and the measures obtained conveyed an average result after several experiments of the same profile.

Figure 14 shows the latency when publishing. For example, a publisher needs 3s to publish 100 events of type DetailRecord. They include the time for marshalling each of the events and the time to put the events into the UDP socket.

Figure 15 shows the global throughput for the same scenario. It takes for instance 5s until a subscriber has received 100 events of type `DetailRecord`. The 5s correspond therefore to the time spent at the publisher side and the subscriber side of the DAC. They include the time for marshalling, remote communication and unmarshalling. These simple measurements allowed us to do draw several preliminary conclusions:

- The complexity of the event type has a heavier impact on the time it takes for a publisher to send events then on a subscriber to receive events. This is not surprising because in the first case, the marshalling time is more significant (there is no inherent cost of remote communication).
- It might look surprising that integers take longer than strings. In this implementation however, everything is converted to strings in the serialization procedure.
- Finally, the overall measures confirm the very fact that nowadays, optimizing marshalling is at least as important as optimizing remote communication.

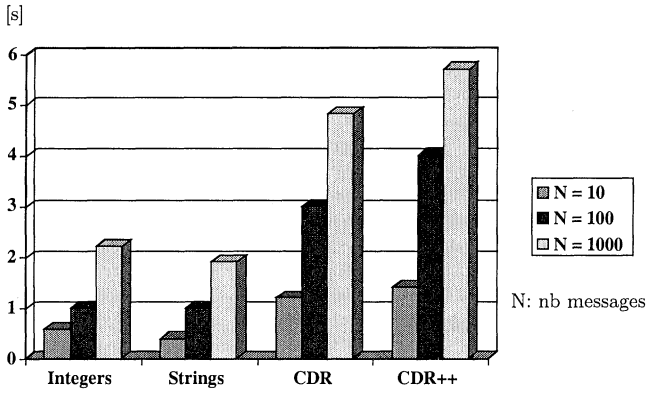


Figure 14: Latency

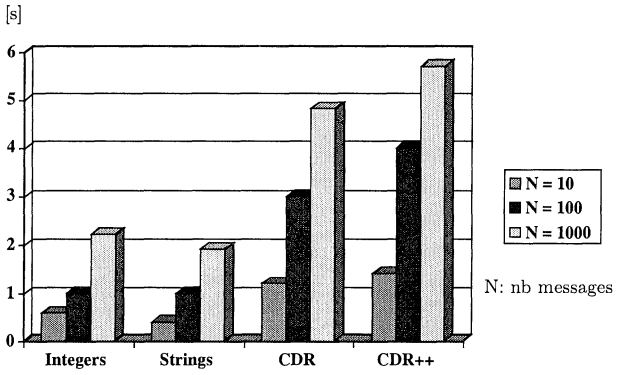


Figure 15: Throughput

8. DISCUSSION

This section discusses two design issues concerning the consistency of queues and their behavior in a hierarchy.

8.1 Scalability issues

A very simplistic implementation of a callback queue consists in using a round robin protocol locally at every producer to determine to which consumer a message should be sent. Pull-style queues are more difficult to implement in a distributed manner than callback queues. Intuitively, a queue accessed through pulling consumers involves more synchronization, not only a synchronization of the queue with each consumer individually, but also

with respect to all consumers. Strong synchronization affects consistency, and our current implementations of the `DABasicQueue` interface are therefore centralized implementations. Note that this is a general issue, which is not a consequence of our approach to queues. Topics are inherently better suited for large scale, exactly because push-oriented solutions require less synchronization than pull-style ones, especially when combined with an all-of-n flavour. For that reason, queues are in contrast to topics deployed more locally.

8.2 Polling Subqueues

With callback queues, the subscription to a queue Q and its subqueues can be realized by multiplexing the subscription to the subqueues of Q. In other terms, a subscription will be generated to each queue Q' in the subhierarchy in which Q is root.

With classical message queues, i.e., used with pulling consumers, the issue of involving subqueues is more delicate, since it additionally introduces synchronization between (sub)queues. Suppose a hierarchy of queues accessed in non-blocking pull-style. Imagine the top-level queue Q is polled, and does not contain any value to return. Hence the subqueues have to be polled. This can be done either in a depth-first order (by first increasing the level), or level by level. If a subqueue Q' is being polled, it could very well happen that a previously empty queue, for instance the root queue Q itself is populated in the meantime. The hierarchical disposition of queues implicitly expresses a priority, which would in this case be violated when returning a value of Q'.

The queues we propose do not prevent this case, i.e., they are implemented according to the convention that the queue hierarchy is checked level by level until the first value is found.

8.3 Blocking Pull

Blocking pull-style interaction on a queue subhierarchy is even harder to realize effectively. A *repetitive polling* of every queue leads to scanning the entire hierarchy one by one for new values, and restarting at the root if the previous run terminated unsuccessfully, i.e., no new value was available in any queue. This might however lead to an important load of the system. On the other hand, when using a blocking pull-style interaction with every subqueue, one might end up in the situation where several messages are obtained. To avoid the drawbacks of both scenarios, we have chosen a specialized protocol, which proceeds by two steps. First, a queue, which has a value can *propose* it to a DAQ proxy which has blocked in behalf of a

consumer. The DAQ proxy may then *accept* the value or refuse it, which means that it has already received a value from another (sub)queue.

9. RELATED WORK

During the last years, the need for large-scale event notification mechanisms has been recognized. Much effort has therefore been invested in this domain, and a multitude of approaches has emerged from academic as well as industrial impulses. We present here the main characteristics of related approaches and we compare them with our Distributed Asynchronous Collections.

9.1 Specifications

In order to integrate the publish/subscribe communication style into existing middleware standards, specifications have been conceived by both the Object Management Group [16, 22] and Sun [21, 23]. For our comparisons, these approaches are the most relevant, since they cover more than one messaging style. Many messaging system vendors implement the API's corresponding to the messaging style(s) their service exploits.

CORBA Event Service. The OMG has specified a CORBA service for publish/subscribe oriented communication, known as the *CORBA Event Service*. The specification is aimed to be general enough to not preclude sub-specifications and various implementations that would match the needs of specific applications. According to the general service specified however, a consumer interacts with an *event channel* expressing thereby an interest in receiving *all* the events from the channel. In other words, filtering of events is done according to the channel names, which basically correspond to topic names. Hierarchical disposition of channels and utomatic subscriptions to subchannels is however not explicited. Event channels are CORBA objects themselves, and in current implementations they are centralized components.

Therefore these engines manifest a strong sensitivity to any component failure, which makes them unsuitable for critical applications.

CORBA Notification Service. The lacks of the event service specification have been realized early, namely concerning QoS and realtime requirements. After the emergence of extended and proprietary approaches aimed at fixing the shortcomings of the event service (e.g., [24]), the OMG

has issued a request for proposal for an augmented service, the *CORBA Notification Service* [22]. A *notification channel* is an event channel with additional functionalities. There is a strong support for typed events, and notions like priority and reliability are explicitly dealt with. The notification service does however not express any distinction between all-of-n and one-of-n semantics.

Java Message Service. The Java Message Service [21] is a specification from Sun. Its goal is to offer a unified Java API around common messaging engines. Its generic nature enables it to conform to a maximum number of existing systems. Certain existing services implement the JMS, but to our knowledge no messaging system has been implemented with the goal to merely support the JMS API directly. Both topic-based publish/subscribe and message queuing are supported, and are represented by different abstraction types (`Topic, Queue`).

Jini. The *Jini Distributed Event Specification* [23] explicitly introduces the notion of event *kind*. Registration of interest indicates the kind of events that is of interest, while a notification indicates an occurrence of that kind of event. One can combine this notion with that of *JavaSpace* [25] to provide support for topic-based publish/subscribe notification. Inspired by Linda [15], a JavaSpace is for example a container of objects that might be shared among various suppliers and consumers. The JavaSpace type is described by a set of operations among which a *read* operation to get a copy of an object from a JavaSpace, and a *notify* operation aimed at alerting some potential consumer object about the presence of some specific object in the JavaSpace. Combined with the Jini Distributed Event interfaces, one can build a publish/subscribe communication scheme where a JavaSpace plays the role of the event channel aimed at broadcasting events (notifications) to a set of subscriber objects. The classification of events is however based on the types of the events, and it is not clear whether it is possible to combine one-of-n semantics with asynchronous notifications.

These standards are based on specifications and it would be interesting to see how one could implement services that comply with these standards using DACs. In contrast to these systems, our DAC programming abstraction focuses on expressing the differences and commonalities of message-oriented interaction styles by subtyping a single basic abstraction.

9.2 Pioneers

The specifications presented above have followed the impulses given by academic research and industrial players. For the sake of brevity we do not

present all existing systems, but only the ones that we believe have had the strongest influence on the evolution of the message-oriented interaction styles we have presented in this chapter.

Topic-Based Systems. Most industrial strength solutions involve topic-based publish/subscribe, like *Smartsockets* [3] or *TIB/Rendezvous* [4].

In Smartsockets, an event channel can accept subscriptions for specific topics. A consumer receives all the event notifications that belong to the topic to which it has subscribed. The topic defines a kind of virtual connector between objects of interest and recipients. If a producer is interested in producing an event on a number of topics or channels, it has to explicitly publish the event on all of them. Event notifications are represented by records, nevertheless custom event types may be defined.

A similar approach was adopted in the development of the TIB/Rendezvous infrastructure. A hierarchical naming model corresponds to the hierarchical organization of the entities of interest. Just as Uniform Resource Locators (URLs) provide a way of locating and accessing Internet resources, a naming scheme is provided to locate and access events of interest. The naming scheme proposed can use wildcards, which allows to subscribe to patterns of topics. TIB/Rendezvous provides a certain degree of fault-tolerance, and makes usage of IP-multicast. Event notifications are composed of a set of typed data fields, including the topic.

Message Queuing. The IBM *MQSeries* [7] is one of the original and definitely the most popular message queuing system. The *Application Messaging Interface (AMI)* spans message queuing as well as publish/subscribe interaction.

MQSeries is a complete framework, in the sense that it is a general solution. It addresses aspects such as security, transactions and especially message storage (*message warehousing*). Message queues are created explicitly through queue managers. MQSeries is based on the notions of message flow and intermediate message brokers, and implements JMS [26].

Most of the above systems offer API's in object-oriented languages like Java. These solutions however did not undergo a fundamentally object-oriented design. Messages are seen as "flat" structures, and are often reduced to a set of system-defined types. We believe that custom message types and classes are very important to ease development. In [27] we extend this idea by introducing *type-based* publish/subscribe, an alternative static subscription scheme based on the message types.

10. CONCLUDING REMARKS

We conclude this work by considerations on current practices in object-oriented distributed programming, and the role of DACs with respect to those practices.

Delusions about distributed objects. It has long been argued that distribution is an implementation issue and that the very well known metaphor of objects as "autonomous entities communicating via message passing" can directly represent the interacting entities of a distributed system. This approach has been conducted by the legitimate desire to provide distribution transparency, i.e., hiding all aspects related to distribution under traditional centralized constructs. One could then reuse, in a distributed context, a centralized program that was designed and implemented without distribution in mind.

As argued in [28, 29, 30] however, distribution transparency is a myth that is both missleading and dangerous. Distributed interactions are inherently unreliable and often introduce a significant latency that is hardly comparable to that of a local interaction. The possibility of partial failures can fundamentally change the semantics of an invocation. High availability and masking of partial failures involves distributed protocols that are usually expensive and hard, if not impossible to implement in the presence of network failures (*partitions*).

Objects vs messages. Message-oriented middleware has emerged during the last few years as a reply to the limitations of the derivatives of the *remote procedure call* (RPC) interaction style (DCOM [31], Java RMI [32], CORBA [33]).

Message-centric approaches, based for instance on the publish/subscribe or message queuing paradigms, are however often claimed to be inherently incompabitle with object-orientation, on the pretext that "objects" cannot really support the requirements of a messaging middleware [34]. This belief has been strongly driven by the argument that objects do communicate through synchronous method invocations which force the interacting parties to be bothcoupled in time and in space.

The argumentation is influenced by the current commercial practices in distributed object-oriented computing, which are mainly based on synchronous remote method invocations. As we convey in this chapter however, decoupling producers and consumers can be made very practical in an object-oriented setting.

DACs: marrying object-orientation and message-oriented middleware. We have been considering an alternative approach where the programmer would be very aware of distribution but where the ugly and complicated aspects of distribution would be encapsulated inside specific abstractions with a well-defined interface. This chapter presents a candidate for such an abstraction: The *Distributed Asynchronous Collection*. It is a simple extension of the well-known collection abstraction. DACs add an asynchronous and distributed flavor to traditional collections [18], and enable to express various forms of message-oriented interaction. In fact, most systems we know about are unwieldy and consider only a limited set of interaction models. DACs are lightweight publish/subscribe abstractions: they can be introduced through a library approach and they express the different message-oriented interaction styles. We are currently integrating new flavors into our framework, for instance through *content-based* filters, which offer a more fine-grained subscription scheme based on properties of message objects.

We believe that our object-oriented view of messaging is a unique compromise between transparency and efficiency. By offering a modular design aligned with different communication semantics, we enforce ease of use without missing performance related issues. We are currently making use of DACs in various practical examples, which are far more complex than the simple chat example presented in this chapter. The objective of investing in several applications is to end up with a stable framework that would for instance extend JGL [17]. The issue of translating operations known from conventional collections to an asynchronous distributed context is however not entirely completed, and certain parts of the API might be affected by future modifications. We also explore specific algorithms to realize efficient matching, especially focusing on the tradeoffs of dynamic and static filter approaches. Finally, we are also focusing on efficient distributed implementations of pull-style queues, especially in combination with a hierarchical deployment.

11. REFERENCES

1. B. Oki and M. Pfluegl and A. Siegel and D. Skeen. *The Information Bus - An Architecture for Extensible Distributed Systems*. In 14th ACM Symposium on Operating System Principles, pp. 58-68, 1993.
2. D. Powell (editor). *Group Communications In Communications of the ACM*, 39(4), pp. 50-97, 1996.
3. Talarian Corporation. *Everything You need to know about Middleware: Mission-Critical Interprocess Communication (White Paper)*. http://www.talarian.com, 1999.
4. TIBCO Inc. http://www.rv.tibco.com/whitepaper.html, 1999.

5. D. Skeen. *Vitria's Publish-Subscribe Architecture: Publish-Subscribe Overview*. http://www.vitria.com, 1999
6. M. Altherr and M. Erzberger and S. Maffeis. *iBus - A Software Bus Middleware for the Java Platform*. In Int. Workshop on Reliable Middleware Systems, pp. 43-53, 1999.
7. B. Blakeley and H. Harris and J.R.T. Lewis. *Messaging and Queuing Using the MQI: Concepts and Analysis*, Design and Development McGraw-Hill, 1995.
8. Digital Equipment Corporation. *DECMessageQ: Introduction to Message Queuing*, 1994
9. BEA Systems Inc. *Reliable Queuing Using BEA Tuxedo: White Paper*. http://www.beasys.com/products/tuxedo/, 2000.
10. Microsoft Corporation. *Microsoft Message Queuing Services*, 1997.
11. D. Schmidt and S. Vinoski. *Overcoming Drawbacks in the OMG Event Service*. In SIGS C++ Report magazine, 1997.
12. E. Gamma and R. Helm and R. Johnson and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
13. M. Hauswirth and M. Jazayeri. *A Component and Communication Model for Push Systems*. In ESEC/FSE 99 - Joint 7th European Software Engineering Conference (ESEC) and, 1999.
14. K.P. Birman. *The Process Group Approach to Reliable Distributed Computing*. In Communications of the ACM, 36(12), pp. 36-53, 1993.
15. D. Gelernter. *Generative Communication in Linda*. In Transactions on Programming Languages and Systems (TOPLAS), ACM, 7(1), pp. 80-112, 1985.
16. OMG. *CORBAservices: Common Object Services Specification*. OMG, 1998
17. ObjectSpace. *JGL-Generic Collection Library*. http://www.objectspace.com/products/jgl/, 1999.
18. J.P. Briot and R. Guerraoui and K.P. Lohr. *Concurrency, Distribution and Parallelism in Object-Oriented Programming*. In ACM Computing Surveys, 30(2), pp. 291-329, 1998.
19. A. Yonezawa and E. Shibayama and T. Takada and Y. Honda. *Modeling and Programming in an Object-Oriented Concurrent Language ABCL/1*. In Object-Oriented Concurrent Programming, pp. 55-89, MIT Press, 1993.
20. D. Caromel. *Towards a Method of Object-Oriented Concurrent Programming*. In Communications of the ACM, 36, pp. 90-102, 1993.
21. M. Happner and R. Burridge and R. Sharma. *Java Message Service*. Sun Microsystems Inc., 1998.
22. OMG. *Notification Service - Joint revised submission*. OMG, 1998.
23. K. Arnold and B. O'Sullivan and R.W. Scheifler and J. Waldo and J. Wollrath. *The Jini Specification*. Addison-Wesley, 1999.
24. T. Harrison and D. Levine and D.C. Schmidt. *The Design and Performance of a Real-Time CORBA Event Service*. In 12th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'97), pp. 184-200, 1997.
25. E. Freeman and S. Hupfer and K. Arnold. *JavaSpaces Principles, Patterns, and Practice*. Addison-Wesley, 1999.
26. IBM. *MQSeries: Using Java*. IBM, 2000.
27. P. Th. Eugster and R. Guerraoui and J. Sventek. *Type-Based Publish/Subscribe*. Technical Report 2000-029, Communication Systems Department, Swiss Federal Institute of technology, 2000.
28. J. Waldo, G. Wyant and A. Wollrath and S. Kendall. *A Note on Distributed Computing*. Sun Microsystems Inc., 1994.
29. D. Lea. *Design for open systems in Java*. In 2nd International Conference on Coordination Models and Languages, 1997.

30. R. Guerraoui. *What object-oriented distributed programming does not have to be and what it may be*. In Informatik, 2, 1999.
31. Microsoft Co. *DCOM Technical Overview (White Paper)*. Microsoft Co., 1999.
32. Sun Microsystems Inc. *Java Remote Method Invocation - Distributed Computing for Java (White Paper)*. Sun Microsystems Inc., 1999.
33. OMG. *The Common Object Request Broker: Architecture and Specification.* OMG, 1999.
34. P. Koenig. *Messages vs. Objects for Application Integration.* In Distributed Computing, 2(3), pp. 44-45, 1999.
35. E. Jul and H. Levy and N. Hutchinson and A. Black. *Fine-grained mobility in the Emerald System*. In ACM Transactions on Computer Systems, 6(1), pp. 109-133, 1998.
36. J. Waldo and G. Wyant and A. Wollrath and S. Kendall. *Events in an RPC Based Distributed System*. Sun Microsystems Laboratories Inc., 1995.
37. D. Rosenblum and A. Wolf. *A Design Framework for Internet-Scale Event Observation and Notification.* In 6th European Software Engineering Conference/ACM SIGSOFT 5th Symposium on the Foundations of Software Engineering, 1997.
38. Sun Microsystems Inc. *The Java Platform 1.2 API Specification*. Sun Microsystems Inc. http://java.sun.com/products/jdk/1.2/, 1999.

Chapter 7

CO-EVOLUTION OF OBJECT-ORIENTED SOFTWARE DESIGN AND IMPLEMENTATION

Theo D'Hondt, Kris De Volder, Kim Mens and Roel Wuyts
Programming Technology Lab, Department of Computer Science, Vrije Universiteit Brussel, Pleinlaan 2, B-1050, Brussels, Belgium. E-mail: {tjdhondt, kdvolder, kimmens, rwuyts}@vub.ac.be, www: http://prog.vub.ac.be

Key words: Software architectures, meta-programming, logic programming.

Abstract: Modern-day software development shows a number of feedback loops between various phases in its life cycle; object-oriented software is particularly prone to this. Whereas descending through the different levels of abstraction is relatively straightforward and well supported by methods and tools, the synthesis of design information from an evolving implementation is far from obvious. This is why in many instances, analysis and design is used to initiate software development while evolution is directly applied to the implementation. Keeping design information synchronized is often reduced to a token activity, the first to be sacrificed in the face of time constraints. In this light, architectural styles are particularly difficult to enforce, since they can, by their very nature, be seen to crosscut an implementation. This contribution reports on a number of experiments to use logic meta-programming (LMP) to augment an implementation with enforceable design concerns, including architectural concerns. LMP is an instance of hybrid language symbiosis, merging a declarative (logic) meta-level language with a standard object-oriented base language. This approach can be used to codify design information as constraints or even as a process for code generation. LMP is an emerging technique, not yet quite out of the lab. However, it has already been shown to be very expressive: it incorporates mechanisms such as pre/post conditions and aspect-oriented programming. We found the promise held by LMP extremely attractive, hence this chapter.

1. INTRODUCTION

Recent times have seen a consolidation of methods and tools for software development in production environments. A good number of de facto standard tools have emerged, not the least of which is the *Unified Modeling Language* in some commercial incarnation. At last, one could say, we have come to know the process by which the design and implementation of a complex piece of software is charted. Objects have been widely accepted; the programming language babel seems more and more controlled by the emergence of Java and previously untamed regions of information technology, such as distribution, co-ordination and persistence, are starting to become daily fare in software applications.

So why is software development still arguably the least predictable of industrial processes? Why can comparable software projects, executed by development teams with comparable skills, not be planned with comparable margins of error? Why is our appreciation of the software development process still flawed, even after the introduction of all these new techniques and tools?

For some time now, grounding the development of software in a programming language has proved not to be scalable. This led to the notion of *software architectures* as a collection of techniques to buttress this development process, particularly in those places where programming languages or tools fail to capture the macroscopic structure of the system that needs to be built. Close to the programming language technology itself, we find the well-understood *framework* approach; at a more abstract level we find techniques built on various kinds of *patterns* and *contracts*.

The latest landslide in this fight for control over software complexity is the emergence of *component* technology. At a time when commercial component toolkits such as *Enterprise Java Beans* are proposed as the solution to our problems, we do well in realizing that the advent of components amounts to an acknowledgement of defeat. In fact, by accepting this technique of decomposition into static components, we have come full circle and reinvented data abstraction. The task of making components co-operate is not any better understood than any of the numerous software building strategies we have taken under consideration these past 20 years.

An important step in understanding this partial failure is the insight that software is fluid. It is in constant evolution under the influence of ever changing conditions; software development is sandwiched between a technology that is evolving at breakneck speed, and requirements that must follow the economic vagaries of modern society. In this, the commercial product called *software* is unique; the closest professional activity to that of software developer is that of composer in 18th century Europe. At that time,

relatively widespread knowledge of harmony or counterpoint made for the necessary skills to use and reuse fragments of sophisticated musical artefacts. For instance, [8] offers insight on how *invention*, a term borrowed from rhetoric, drives composition according to a process which bears a striking resemblance to building complex computer applications. Unfortunately, equivalent skills needed to master a software artefact are today in far more limited supply than 250 years ago.

This contribution is a synthesis of recent work performed by various people within our lab in addressing this need for more control over the evolution of software. Although in the past, a significant amount of work focussed on the need to *document* evolution and build *conflict detection tools* [1, 14], to *recover* architectural information from implementations [6] and to *formalize* the evolution process [13], we will concentrate here on an emerging approach for *steering* evolution. This is very recent work and as such has only resulted in experiments and prototypes. We feel however that it is sufficiently mature and promising to be presented here as a whole. In the bibliography we limit ourselves to a number of key documents[1] describing these activities; these in turn contain a much more comprehensive list of references.

We have chosen to use the term *co-evolution*, implying that managing evolution requires the synchronization between different layers (or views) in the software development process. We will therefore dedicate the next section to an analysis of this statement. Next, we propose a concept called *Logic Meta Programming* (or LMP for short) as a development framework in which to express and enforce this synchronization process. Another section of this chapter will be used to introduce LMP and to situate it in the broader context of software development support. Finally, several experiments with LMP will be presented in evidence of its applicability. We will discuss using LMP as a medium for supporting *aspect oriented programming*, for enforcing *architectural concerns* in an object oriented programming environment and to express constraints on the protocol between a collection of interacting software components. This is by no means a complete coverage of LMP, nor even of the experiments conducted at our lab; we feel however, that it provides sufficient insight in the applicability of LMP to the co-evolution of software, while avoiding exposing the reader to too much detail.

[1] Available via http://prog.vub.ac.be

2. SYNCHRONIZING DESIGN AND IMPLEMENTATION

Currently accepted procedures in the development of software involve adopting several views. In descending order of abstraction one encounters *requirements capture*, *analysis*, *design*, *implementation*, *documentation* and *maintenance*. Not quite by accident, this also happens to constitute an ordering according to increasing level of detail, albeit not a continuous one. There is in fact a kind of watershed between design and implementation, which commits the developer to a level of detail that is very hard to reverse.

Consider an example describing a simple management hierarchy. At best this is captured at the class diagram level by an arity constraint, although the more subtle aspects such as the required absence of cycles in the hierarchy graph can only be expressed by an informal annotation:

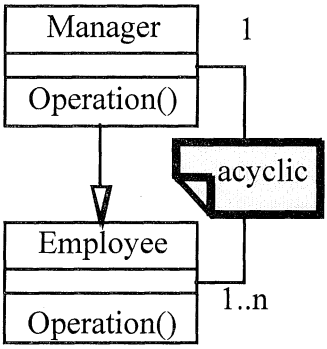


Figure 1: A manager/employee hierarchy

It can be seen that as we transit from the original requirements to the design, we replace abstract concepts by more concrete ones. The same holds for the implementation: in our simple example the arity constraint might be replaced by a precondition in a mutator function while the acyclicity constraint, if implemented at all, gives rise to some consistency maintenance code. On the other hand, we see that this decrease of abstraction is compensated by an increase in the level and amount of detail.

This observation holds in general and will be viewed as trivial by most software developers. However, we do well in analyzing this transition from abstraction to detail as we descend through the various levels in the lifecycle of a software application. Typically, the amount of energy that needs to be applied increases with the level of detail; so does the need for technical skills. This generally makes an implementation artefact more valuable than a

design artefact. Also, any ultimate defect in respecting the original requirements is detected at the lowest level, i.e. the implementation.

Initially, the development of a software application is achieved by this progression through the various abstractions: requirements, analysis, design and implementation. However, once the implementation has reached the production stage, the tangible aspects of the prior stages are at best used as documentation in order to boost understanding of the actual code; at worst they become obsolete. This is a well-known phenomenon: under the pressure to bring software to market in the face of competition, or to correct flaws under the threat of contractual penalties, the management of evolving software all too often degenerates into updating implementations. Various directions have been explored to improve this situation: in general they imply some re- engineering activity applied to evolving implementations in order to extract abstractions and update e.g. design documentation. Hardly anyone uses an approach where design concerns drive the implementation process; programming environments that explicitly constrain the developer to design decisions are hard to find. Popular languages like Java evolve, but they evolve towards a more sophisticated type system: boosting genericity is a technical issue and hardly qualifies as support for e.g. architectural concerns.

It is our conjecture that during the development process, the concretization of abstract concerns should not consist of some kind of erosion. On the contrary, any relevant feature should be kept available in any of the ulterior phases. We will in particular concentrate on the synchronization between *design* and *implementation*. For the sake of this discussion we will discard requirements that cannot be expressed as explicit design directives. Our ambition is to augment an implementation such that it becomes a strict superset of its design; design can be extracted from an implementation by ignoring details; design can be interpreted by the programming environment and therefore enforced. We propose an approach called *Logic Meta Programming*, (or LMP for short) which will be described in the now following section.

Consider as an example a change in the manager/employee example where a decision to introduce *workforce pooling* results in the arity constraint to be changed into:

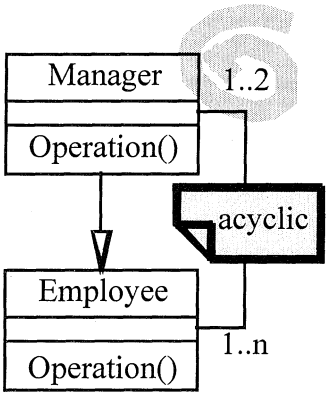


Figure 2: A manager/employee hierarchy with workforce pooling

The program code will probably need to be changed significantly with hardly an explicit link to the original arity constraint. We would prefer it to be explicitly present in the implementation as some kind of enforceable declaration, formatted in the proposed LMP-paradigm.

3. LOGIC META PROGRAMMING

Logic Meta Programming, or LMP for short, is the name we use for a particular flavor of multi-paradigm programming. The starting point for LMP is an existing programming environment that is particularly suited for engineering large software systems. In this contribution, we have limited ourselves to a Java-based environment and to a Smalltalk-based environment. Next, we augment this environment by a declarative meta layer of a very particular nature. In the case of Java, i.e. a statically typed language, this meta layer might be implemented as a pre-processor or even an extension of the Java compiler itself. In the case of Smalltalk, it requires the addition of a number of classes to the standard Smalltalk hierarchy. We are interested in a declarative approach; it seems intuitively clear that design information, and in particular architectural concerns, are best expressed as constraints or rules. Logic programming has long been identified as very suited to meta programming and language processing in general; see [4] for related publications.

The acyclicity constraint from the manager-employee example on the previous page seems to indicate[2] the need for unification as an enforcement

[2] Imagine for instance a tool to enumerate al cyclic calling graphs.

strategy. Anyway, we would like as much power on our side as possible, at least initially. We are not concerned with performance issues at this stage; neither do we intend to explore all avenues of declarative programming. For historical reasons, we concentrate on a Prolog-derivative for our logic meta language; its power and its capacity to support multi-way queries seem particularly attractive at this point. Finally, we make the symbiosis between the two paradigms explicit by allowing base-level programs to be expressed as terms, facts or rules in the meta-level; we will refer to this as a *representational mapping.*

Consider the simple Java class in figure 3: it implements an array of integers. Following it in figure 4 the original class has been embedded in a meta-declaration using a representational mapping. The notation is fairly crude and to clarify it somewhat elements from the meta-program have been highlighted. Notice that the original element type was replaced by a logic variable.

```
class Array {
 private int[] contents;
 Array(int sz) {
 contents = new int[sz];
 }
 int getAt(int i) {
 return contents[i];
 }
 void setAt(int i, int e) {
 contents[i] = e;
 }
}
```

Figure 3: A Java array

This is an example of using LMP for code-generation; it was explored in [7] under the exotic name *TyRuBa*. A proper query substituting `int` for **`?El`** in figure 4 would produce figure 3. Actually, figure 4 is a simple example of how LMP can be used to introduce parametric types.

```
Class(Array<?El>, {
 private ?El[] contents;
 Array<?El>(int sz) {
 contents = new ?El[sz];
 }
 ?El getAt(int i) {
 return contents[i];
 }
 void setAt(int i, ?El e) {
 contents[i] = e;
 }
});
```

Figure 4: A generic Java array

A totally different way to view LMP is introduced in [12] as the *Smalltalk Open Unification Language* (SOUL). This approach actually applies constraints specified at the meta level to the base level program. The representational mapping is based on the presence of predicates that give access to syntactic elements belonging to the base level.

```
Rule transitive(?c1,?c2,?tried) if
  member(uses(?c1,?c2),?tried), !.
Rule transitive(?c1,?c2,?tried) if
  uses(?c1,?c2), !.
Rule transitive(?c1,?c2,?tried) if
  uses(?c1,?c3),
  transitive(?c3,?c2,<uses(?c1,?c3)|?tried>).
Rule cyclic(?c) if
  transitive(?c,?c,<>).
Rule uses(?c1,?c2) if
  class(?c1),
  method(?c1,?m),
  calls(?m,c3).
```

Figure 5: A circularity test

In the above example we assume the availability of predicates `class`, `method` and `calls` to access the structure of a base program. The `cyclic` rule verifies whether there is a transitive calling relationship from a class to

itself. This rule in turn could be used to enforce the acyclicity constraint from the manager-employee example.

4. *LMP* AND ASPECT ORIENTED PROGRAMMING

In [7] an LMP framework is proposed that supports sophisticated type systems for statically typed programming languages such as Java. This framework, called *TyRuBa*, turns a type system into a computationally complete environment and allows a programmer to specify the static structure of a program as a set of logical propositions. In one of the next sections we will report on an experiment to use *TyRuBa* as a system to describe software architectures with. In this section we will build on the relationship between LMP and *Aspect Oriented Programming* (or AOP for short). We refer to [4] for an extended bibliography; suffice it to say that AOP is concerned with the production of software as a result of a *weaving* process. The weaver is an AOP-related tool that is capable of merging aspects of a software application, each of them described in a specific aspect language.

In [4] it is proposed that LMP may well function as an aspect-oriented programming environment. As evidence for this, a well-known case for AOP (synchronization of co-operating processes using an aspect language called COOL) is expressed in *TyRyBa*. An important conclusion drawn from this experiment is the fact that a general-purpose framework, in casu LMP, can be used to host aspect programs; hitherto, aspect languages were specific to the aspect under consideration.

In deference to the subject of this contribution, we will not concentrate on technical applications of AOP; instead we will consider an interesting application of AOP involving design as much as implementation. In [3] the idea is launched that *domain knowledge* might well constitute an aspect in the AOP sense. Separating some problem into its domain aspect and its implementation aspect by describing them in a/some aspect language and then producing a piece of software by applying a weaver seems a very attractive approach.

Moreover, it seems to fit very well with the concept of software co-evolution introduced earlier on.

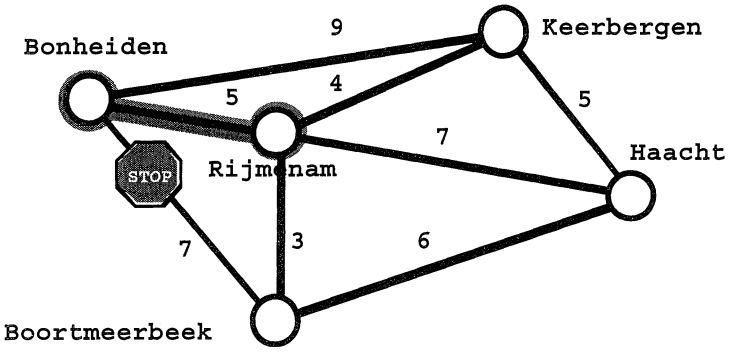


Figure 6: A shortest path problem

Figure 6 represents the test case proposed in [3] to explore this idea. The example is taken from a GIS-application, involving a mix of a conventional algorithm to compute a shortest path, and the specifics of the domain, which allow us to improve the basic algorithm.

```
branchAndBoundFrom: start to: stop
  | bound |
  bound := 999999999.
  self traverseBlock:
    [:node :sum |
     node free ifTrue:
       [sum < bound ifTrue:
         [node = stop ifTrue:
           [bound := sum] ifFalse:
             [self branch: node sum: sum]]]].
  self traverseBlock value: start value: 0.
  ^bound
```

Figure 7: The branch-and-bound program

In order to keep things as simple as possible, we consider an elementary *branch-and-bound* strategy. In figure 7 this is implemented using an auxiliary `branch:` method in order to fix the sequence in which branches are selected.

```
branch: node sum: sum
  node free: false.
  node edges do:
    [:edge | self traverseBlock
               value: edge next
               value: sum + edge distance].
  node free: true
```

Figure 8: Fixing the selection order

Figure 8 contains a possible implementation for `branch:` and it contains an enumeration of all possible edges leaving a node. However, the message `edges` is no longer resolved by the base program, but by a query in the logic meta program containing the knowledge about this particular domain.

```
Fact city(Rijmenam)
Fact city(Boortmeerbeek) ...
Fact road(city(Rijmenam),city(Boortmeerbeek),[3])
Fact road(city(Keerbergen),city(Rijmenam),[4])
...
Fact prohibitedManoeuvre(city(Rijmenam),
                         city(Bonheiden))
Rule roads(?current,?newResult)if
  findall(road(?current,?next,?distance),
  road(?current,?next,?distance),?result)
  privateRoads(?current,?result,?newResult)
Rule privateRoads(?current,?result,?newResult)if
  prohibitedManoeuvre(?current,?next),
  removeRoad(?result road(?current,?next,
                          ?distance),?newResult)
Fact privateRoads(?current,?result,?result)
```

Figure 9: The domain knowledge

The particular flavor of LMP we use here is the *Smalltalk Open Unification Language* mentioned earlier on. The rule needed to compute the edges of a node would look something like this:

```
Rule edges(?node, ?result) if
     equals(?name, [?node name]),
     roads(city(?name), ?result).
```

In [4] an explanation is given of how the base program and the meta program communicate. The basic idea is to effect a kind of linguistic symbiosis (see e.g. [11]) based on a two-way reification of language entities; in the case of SOUL this amounts to wrapping Smalltalk objects inside Prolog facts and vice versa. For example, the code `[?node name]` in the `edges` rule is reified Smalltalk code sending a unary message `name` to retrieve the name from the node currently associated with the variable; it returns a string representing the name. A number of technical issues need to be resolved still; in particular SOUL would seem to lack in reflective power and needs to be extended with a number of reification operators. Also, the proposed test case would seem to border on the trivial. On the other hand, it shows that there is at least a lower bound to a category of problems that can be non-trivially decomposed into a *domain* part and an *implementation* part using AOP. An interesting research topic concerns the charting of this category and the development of tangible procedures to perform the related decomposition. The proposed LMP approach seems at the very least attractive enough to function as a vehicle for this research.

5. *LMP* AND SOFTWARE ARCHITECTURES

Software architectures are concerned with the abstract structure of some software application in terms of building blocks, and the interaction between them. Starting from this fairly broad statement, a number of more specific — and sometimes competing—definitions have been proposed. In [6] it is suggested that software building blocks need not be explicitly linked but may equally well be *classified*. Classification in its simplest form implies that all software entities are *tagged*; together with the possibility to *nest* classifications, this results in a very interesting view on architecture; its simplicity belies the power and expressiveness that was demonstrated in [6].

In [10] LMP is explored as a framework in which to express software architectures using this classification approach. In particular, virtual classifications are proposed, i.e. classification is not limited to simple tagging, but allows every software entity to be associated with a computational classifier. This could e.g. be a logical predicate that is evaluated every time a query is launched, its behavior depending on values submitted by the query.

SOUL is proposed as both the target as the medium for this study: it is used as a kind of architectural description language and it is applied to the architecture of SOUL itself.

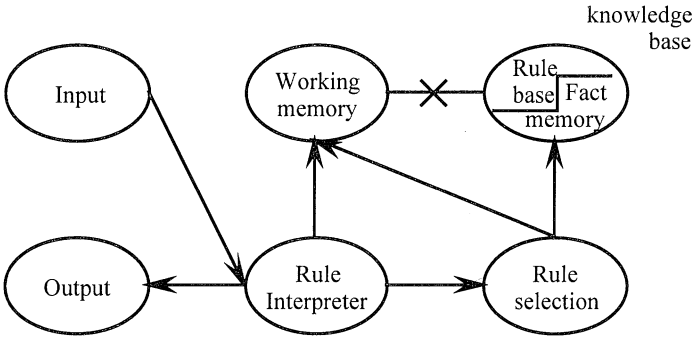


Figure 10: The SOUL rule base architecture

The kernel of SOUL is a logic query interpreter with the above architecture. This architecture is representative for rule bases in general; moreover it is sufficiently challenging to be used as a case study.

This architecture is obtained by means of classification. SOUL being implemented in Smalltalk, *methods* and *classes* are considered as building blocks, and classification is initially limited to a *uses* and *creates* relationship between these entities. For instance, the *working memory* is simple to define: it contains all classes that derive form a root class that specifies the generic structure of variable–value bindings. A more challenging example from [10] is the rule that specifies how methods are classified as belonging to the query interpreter:

```
Rule methodIsClassifiedAs(?Method,queryInterpreter) if
  classImplements([SOULQuery],
                  [#interpret:repository:],?M),
  reaches(?M,?Method).
```

Bracketed terms are wrapped Smalltalk identifiers: `SOULQuery` is the class containing the `interpret:repository:` method that launches interpretation. The predicate `reaches` is similar to `transitive` in figure 5; it verifies that `?Method` belongs to the transitive closure of `?M`.

This classification crosscuts the static structure of the SOUL implementation and could not have been obtained through a simple hierarchical approach. It illustrates the true power of a computationally

complete language in which to express an architecture in terms of classification.

In [10] LMP is also used to express the connectors in figure 9 between the components defined by the various classifications. In fact, the *uses* and *creates* relationships are combined with *universal* and *existential* cardinality constraints to define a limited family of connectors; this results in a specification of the architecture of figure 10 as a ten–line SOUL fact. A definition for the *uses* and *creates* relationships for each of the kinds of implementation artefacts allows this fact to be used to perform *conformance checking* of any SOUL implementation.

This section described a second interesting experiment in using LMP to link the implementation of a software artefact to its design. Although the test case is small-scale and the sophistication of the connectors is limited, the results are promising. [10] claim that it is possible to use this approach to define or even extract *architectural patterns*, which certainly illustrates the power of the proposed formalism. On the downside, performance is an issue requiring a lot of attention to make the LMP approach to software architectures a truly scalable one.

6. *LMP* AND SOFTWARE COMPONENTS

Software components and software architectures are notions that are strongly linked. However, this section is fundamentally different from the previous one. We are no longer interested in the declaration and enforcement of architectural rules and conventions, but in a *composer* environment. In this section we are much more tool-minded, and we want to investigate how LMP can help us drive the process of assembling components. This is of course a major concern for everyone involved in software architectures; again we refer to [9, 10] for a more comprehensive bibliography.

In [9] an experimental generic builder tool is described and applied to the popular *Java Beans* component model. Its architecture is as follows:

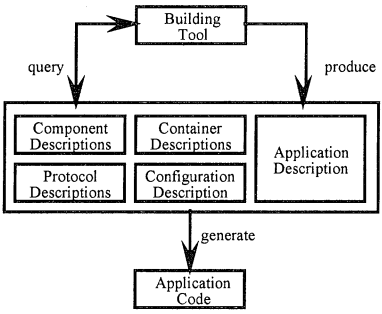


Figure 11: Component builder architecture

The builder tool is supposed to guide a user in establishing a description of the application, and consequently generate the *Java Beans* application code. In order to do so, the tool must have access to a repository of descriptions of the various elements that constitute our component model.

The *TyRuBa* approach from [7] to LMP is used to establish a set of facts and rules to define notions about *parts* and *containers*, and how to link them using *connectors*. Using these, the description of the application becomes a meta-program similar to figure 4.

The component builder architecture provides the mechanism for describing *components*. A *Java Bean* for instance, is defined in terms of its properties, public methods and events. A set of facts allow the specification for accessor/mutator methods (in the case of properties), public methods and listener methods (in the case of events). To get a feeling for the way these are formatted, we include an example from [9] that specifies the registration of an event listener:

```
Feature1(OurButton,
         method<void,add<Action<Listener>>,
         [Action<Listener>]>).
```

The expressions bracketed by < > are compound terms and are used to support the representational mapping of Java in *TyRuBa*.

Next, it is necessary to describe *containers*; a container is a composite application (e.g. an *applet*) and it contains (generates) code to initialize the *parts*. A part is a description of how a component is used inside a container and it contains the specifics of the initialization code. The *configuration* specifies how the writeable properties of parts belonging to a container are set. The *connection protocols* specify how the parts inside a container interact. We refer to [9] for an extended example of the proposed component

model architecture using *TyRuBa*; it would take up too much space to do so here.

The major contribution of [9] is the proof (by construction) that it is possible to separate a builder tool from a component model. It is yet another illustration of the fact that LMP can actively assist in expressing abstract and concrete aspects of a software application within the same framework. Although the proposed builder hardly qualifies as more than a prototype, it indicates an interesting avenue of research. In the spirit of Smalltalk's *Model View Controller*, sophisticated interactive tools can be seen to be separable in independent sections. However, contrary to MVC, a relatively simple framework approach is hardly ever sufficient and more sophisticated techniques are called for. It would seem that using LMP gives at least a partial reply to this concern.

7. CONCLUSION

This contribution is a first synthesis of work that has been going on in our lab these past couple of years related to declarative meta level programming. In particular, it covers several endeavors to marry a logic meta-program to a base program developed in a standard object-oriented programming language. In all cases the major concern was to effect a linguistic symbiosis in order to have the base program query the meta level to resolve issues at an abstract level, and to have the meta program access the structure of the base program.

This synthesis constitutes a push towards research in managing the co-evolution of design and implementation of software applications. We advocate the need to express design as closely integrated with the implementation and we propose *logic meta programming* as a possible way to effect a bi-directional link between the two. Our conjecture is that design becomes verifiable and possibly enforceable if it is properly expressed as a logic meta program. We explicitly address cases where software is subject to evolution, and where synchronization between design and implementation is an issue. We are not only interested in the impact of a design change on the derived implementation; we are expressly concerned with the (unfortunately realistic) situation where an implementation is updated and the design needs to be brought in line with these changes.

Given the proper framework, a logic meta program expressing some design can assist a programming environment in constraining a programmer to abstract design rules that are visible at the level of the programming language only in terms of their constituent implementation details. An

inkling that this might be feasible is given by the prototype *Java Beans* application builder.

Logic meta programming can also assist in separating the development of a software application into domain concerns and implementation concerns. Given that evolution of software at the implementation level is often inspired by implementation issues, this uncoupling of concerns might significantly and positively impact the proposed process of co-evolution. As was illustrated earlier on, the capacity of logic meta programming to express different aspect programs in one unifying framework seems far too attractive to ignore.

Possibly the most interesting direction described in this contribution is the management of co-evolution through virtual classifications. Logic meta programming seems extremely well suited to the annotation of object-oriented software with queries that are automatically triggered when some element is changed; depending on the content of the rule base, these queries can ensure the synchronization between the various abstraction layers. Experiments with a simple tagging strategy have shown significant promise; opening up this strategy to query-based classification should give us the key to controlling the level of detail that co-evolution should respect.

There are of course a number of unresolved issues; this is after all a research topic barely out of the bud. A major concern is one of *efficiency* and *performance*. It would seem that depending on the degree of support offered by the proposed approach, various flavors of declarative meta programming with various performance ratings should be considered. Obviously, the vast field of research in declarative languages can function as an inspiration. Next, other levels in the lifecycle of software should be considered. In particular, expressing requirements (particularly the non-functional ones) in an LMP framework could prove to be a fascinating and rewarding research topic.

Finally, there is an enormous need to validate these ideas in a production setting. Although this has been initiated on a limited scale, many more experiments are needed. On the other hand, there is an even greater need for *comprehensive* software lifecycle management methods and tools. Steering co-evolution between design and implementation using logic meta programming seems to be an interesting step in this direction.

8. REFERENCES

1. C. Lucas. *Documenting Reuse and Evolution with Reuse Contracts*. PhD dissertation, Vrije Universiteit Brussel, 1997.

2. M. D'Hondt and T. D'Hondt. *Is domain knowledge an aspect?*. Proceedings of the ECOOP99 Aspect Oriented Programming Workshop, 1999.
3. M. D'Hondt, W. De Meuter, and R. Wuyts. *Using Reflective Programming to Describe Domain Knowledge as an Aspect*. Proceedings of GCSE'99, 1999.
4. K. De Volder, and T. D'Hondt. *Aspect-Oriented Logic Meta Programming*. Proceedings of Reflection'99, 1999.
5. J. Brichau. *Syntactic Abstractions for Logic Meta Programs, or vice-versa*. Draft publication, 1999.
6. K. De Hondt. *A Novel Approach to Architectural Recovery in Evolving Object-Oriented Systems*. PhD dissertation, Vrije Universiteit Brussel, 1998.
7. K. De Volder. *Type-Oriented Logic Meta Programming*. PhD dissertation, Vrije Universiteit Brussel, 1998.
8. L. Dreyfus. *Bach and the Patterns of Invention*. Harvard University Press, 1996.
9. M. J. Presso. *Generic Component Architecture Using Meta-Level Protocol Descriptions*. Master's dissertation, Vrije Universiteit Brussel, 1999.
10. K. Mens, R. Wuyts and T. D'Hondt. *Declaratively Codifying Software Architectures Using Virtual Software Classifications*. Proceeding of TOOLS Europe'99, 1999.
11. P. Steyaert. *Open Design of Object-Oriented Languages, a Foundation for Specialisable Reflective Language Frameworks*. PhD dissertation, Vrije Universiteit Brussel, 1994.
12. R. Wuyts. *Declarative reasoning about the structure of object-oriented systems*. Proceedings of TOOLS'98 USA, 1998.
13. T. Mens. *A Formal Foundation for Object-Oriented Evolution*. PhD dissertation, Vrije Universiteit Brussel, 1999.
14. P. Steyaert, C. Lucas, K. Mens and T. D'Hondt. *Reuse Contracts: Managing the Evolution of Reusable Assets*. Proceedings of OOPSLA, ACM SIGPLAN Notices number 31(10), pp. 268-285, 1996.

Chapter 8

DERIVING DESIGN ALTERNATIVES BASED ON QUALITY FACTORS

Mehmet Akşit and Bedir Tekinerdoğan
TRESE Group, Department of Computer Science, University of Twente, postbox 217, 7500 AE, Enschede, The Netherlands. email: {aksit, bedir}@cs.utwente.nl, www: http://trese.cs.utwente.nl

Keywords: modeling design spaces, design alternatives, balancing quality factors

Abstract: Software is rarely designed for ultimate adaptability or performance but rather it is a compromise of multiple considerations. At almost every stage of the software development lifecycle, software engineers have to cope with various design alternatives. Current object-oriented design practices, however, rely mainly on the intrinsic quality factors of the object-oriented abstractions rather than considering quality factors as explicit design concerns. It is considered important to support software engineers in identifying, comparing and selecting the alternatives using quality factors such as adaptability and performance. This chapter introduces a new technique to depict, compare and select among the design alternatives, based on their adaptability and time performance factors. This technique is formally specified and implemented by a number of tools.

1. INTRODUCTION

Software development methods [6][13] provide a set of heuristic rules to guide software engineers to analyze, design and implement software systems. Although heuristic rules can be quite helpful in developing high quality software systems, it is generally difficult for software engineers to identify, compare and prioritize the design alternatives.

Software is rarely designed for ultimate adaptability or performance but rather it is a compromise of multiple considerations. At almost every stage of

the software development lifecycle, software engineers have to cope with various design alternatives. Of course while defining object models, software engineers apply their knowledge and experience. They generally compare the alternatives based on their intuition. This process, however, is rather implicit instead of explicit.

This chapter introduces a new technique called Design Algebra to depict, compare and select the alternatives of a design, based on the adaptability and time performance factors. The software engineer can identify the alternatives at various phases of the development process, and make explicit decisions. This technique is formally specified and implemented by a set of tools.

This chapter is organized as follows. The next section introduces an example and explains the problems addressed in this chapter. Section 3 presents a process for selecting the design alternatives based on the adaptability factors. Section 4 shows techniques to determine the probabilistic time performance factors of the design alternatives. Balancing the adaptability and performance factors is explained in section 5. The related work is presented in section 6. Section 7 discusses the usefulness of the proposed approach. Finally, section 8 gives conclusions.

2. THE PROBLEM STATEMENT

We will use a simple working example throughout the chapter. Section 2.1 presents the example, and section 2.2 explains the problems addressed in this chapter using this example.

2.1 An Illustrative Example: Collection Classes

Assume that we would like to design a set of collection classes, such as *LinkedList*, *OrderedCollection* and *Array* to be a part of an object-oriented library. These classes should provide the necessary operations to read and write the elements stored in collection objects. Further, we would like to define a sorting algorithm to order the items stored in collection objects according to a certain criterion.

Now assume that we analyze this requirement specification using an object-oriented method, such as OMT [13]. OMT defines a set of rules and a process to identify the necessary classes, associations, aggregations, attributes, inheritance relations and operations[1]. Applying OMT will possibly result in the class diagram shown in Figure 1. Here classes *Library*,

[1] In this chapter our focus is not OMT but we use OMT as an example of object-oriented software development methods.

LinkedList, *OrderedCollection*, and *Array* are identified by searching for the nouns in the requirement specification. Since these classes share the same abstract behavior, an abstract class *Collection* is introduced, which declares the necessary operations for all its subclasses. The identification of the aggregation relation and the operations *read*, *write* and *sort*, and the attribute *collectionItems* are derived from the requirement specification using the heuristics of the OMT method.

While implementing the class diagram shown in Figure 1, the software engineer has to choose among several options. First, a suitable sorting algorithm has to be selected. Second, the structure of the attribute *collectionItems* must be determined. This structure will mainly depend on the subclass. For example, *LinkedList* will likely have a different attribute structure than *Array*. The operations *read* and *write* and if necessary other operations must be defined and implemented when the sorting algorithm and attribute structure are known. Further, the software engineer must determine if the sorting operation must be implemented in class *Collection* or in the subclasses. If the sorting operation is an abstract (virtual) operation in class *Collection*, then it has to be decided whether some parts of the algorithm must be implemented in the subclasses. Clearly, the class diagram shown in Figure 1 can be implemented in many different ways.

Dealing with the alternatives is not only a concern of implementation. The class diagram shown in Figure 1 is just an example solution for the problem. For example, the sorting operation might be defined as a part object of class *Collection*. This would allow changing the sorting operation at run-time. One might also prefer to change part of the sorting algorithm, for example the sorting criteria (e.g. the Strategy design pattern). There are, of course, a considerable number of design alternatives, depending on the granularity of the required changes, and whether these changes must be realized at compile-time or run-time.

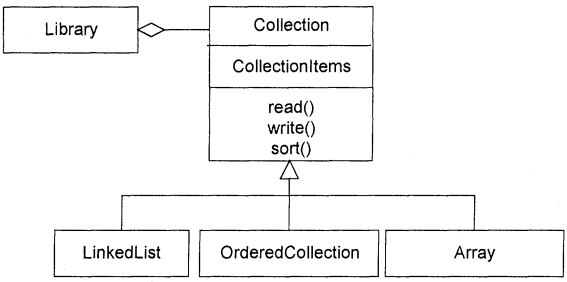


Figure 1: Class diagram of the collection classes

2.2 Problem Description

As illustrated in the previous section, the *design space* of the collection classes is determined by many issues, such as data structures, sorting algorithms, and object-oriented modeling techniques. Although the presence of many alternative solutions indicates the richness of the object-oriented approach, the lack of tool support to compare the alternatives increases the complexity of the analysis and design process.

Consider the object model as shown in Figure 1. We notice two major problems in defining this object model using an object-oriented method like OMT. First, there is no explicit support for identifying the possible design alternatives. Second, although software engineers may prefer to compare the design alternatives based on certain quality factors such as adaptability, performance and reusability, there are no explicit rules to compare the alternatives. Of course while defining object models, software engineers apply their knowledge and experience. They generally compare the alternatives based on their intuition. This process, however, is rather implicit instead of explicit.

Software is rarely designed for ultimate adaptability, performance or reusability but rather it is a compromise of multiple considerations. In general there are many correct solutions for the same problem. Even in the simple case of sorting items in collection objects, one may identify many alternative designs, which will differ with respect to adaptability, performance and reusability factors. Providing ultimate adaptability may create too much run-time overhead. Aiming at the fastest implementation may result in unnecessarily rigid software. Aiming at the most reusable software may introduce redundant abstractions for a given problem. Software engineers must be able to explicitly compare, evaluate and decide between various alternatives based on the relative importance of the quality factors.

3. DESIGNING FOR ADAPTABILITY

In this section we will introduce a process to transform a requirement specification into adaptable object-oriented models. The objective of this process is to gradually introduce domain, design and implementation knowledge into the requirement specification, while selecting the alternatives based on their adaptability factors. This process is formally specified and implemented in a set of tools. We will apply these techniques to the example problem presented in the previous section.

The design process for adaptability consists of the following phases:

Finding the concepts in the requirement specification (section 3.1): This is a necessary step for every software development activity. The output of this phase is a set of concepts.

1. Finding the concepts using domain analysis (section 3.2): In this phase the fundamental abstractions are searched within the context of the solution domain. The objective of this phase is to enrich the concepts obtained from the requirement specification with the concepts that are considered essential in the solution domain.
2. Identification of the adaptable concepts (section 3.3): In this phase the software engineer decides which concepts must be adaptable or fixed. The software engineer may also consider various alternatives and assign adaptability degrees to the alternatives. The purpose of this phase is to make the software engineer conscious about his/her decisions with respect to the adaptability characteristics of the models that he/she develops.
3. Identification of the object-oriented abstractions (section 3.4): The adaptable or fixed concepts delivered from the previous phase are classified according to the object-oriented abstraction techniques. The result of this phase is a consciously selected set of object-oriented abstractions with well-defined adaptability characteristics.
4. Identification of the object-oriented relations (section 3.5): This phase aims at identifying the relations among the identified concepts. The result of this phase is a set of object-oriented relations that satisfy the adaptability requirements.

The total result of this process is a set of alternative object-oriented models that implement the requirement specification. These models may be ordered according to the desired adaptability characteristics. Using this ordering, the software engineer may consciously select one among them. The software engineer may also compare the alternative models both from the adaptability and performance viewpoints. This will be explained in section 5.

3.1 Finding the Concepts in the Requirement Specification

To develop high quality software, it is necessary for any software development method that there is a well-defined requirement specification and sufficient knowledge available about the problem domain. We define a model M as a tuple consisting of a set of concepts and a set of relations

among these concepts. Let us now assume that $M_{Library}$ is a requirement specification model of the collection library example given in section 2.1:

$$M_{Library} = (C_{Library}, R_{Library}) \qquad (F1)$$

$M_{Library}$ is represented by the sets $C_{Library}$ and $R_{Library}$, which correspond to the concepts and relations of the requirement specification, respectively. After analyzing the problem description, we identify the following set of concepts in $C_{Library}$:

$$C_{Library} = (Library, Collection, LinkedList, OrderedCollection, Array, collectionItems, sort, read, write) \qquad (F2)$$

Note that these elements correspond to the elements of the object model shown in Figure 1. However, in (F2) we have not yet make any assumption about the types of object-oriented abstractions that represent these concepts. The relation set $R_{Library}$ will be considered in section 3.5.

3.2 Finding the Concepts using Domain Analysis

To identify the fundamental abstractions we will now analyze the domain of the problem. This commonly involves collecting the related information from various sources, and detecting the commonalties among them through comparison. These common abstractions generally correspond to the fundamental concepts in that domain[2]. The software engineer is responsible for combining the entities obtained from the requirement specification and the concepts from the background knowledge.

We may discover the concepts of the sorting domain by comparing some well-known sorting algorithms. A number of these sorting algorithms is given in [14]. After comparing these algorithms, we can see that they all share the following 5 concepts: the algorithm type, the range of the sorting process, reading and writing items in the collection, and the criterion to compare the items.

The algorithm type basically defines the control-flow of the sorting process, and is used in the literature to distinguish the sorting techniques from each other. In [14], for example, *selection*, *insertion*, and *bubble* sort algorithms are presented. These sorting algorithms perform different with respect to various factors. For example, the *bubble sort* algorithm is very efficient if the collection is almost sorted. The range of the sorting process

[2] To identify the fundamental abstractions of a software system, in chapter 5 of this book a synthesis-based approach is presented. Our approach here follows the synthesis-based approach. The reader may also refer to the various domain analysis methods presented in the literature [3] [3].

defines which items in a collection must be sorted. Although, in [14], the range in all examples is set to the full size, after reading the motivation of the sorting algorithms, we decided to introduce the range as a concept. This allows modeling a partial sorting process. To be able to compare the items in a collection, the items must be read, and to change their order, they must be re-written in the collection. Obviously, sorting must be based on a certain criterion. In a simple case, numbers can be sorted by comparing their magnitudes, or in a more complicated case, items can be compared with each other based on certain policy, such as the arrival date, price, etc. Based on these observations, we define the fundamental concepts of the sorting domain as follows:

$$C_{Sort} = (AlgT, RN, RD, WR, CR) \qquad (F3)$$

Here, C_{Sort} represents a set of concepts of the sorting domain where *AlgT, RN, RD, WR* and *CR* are the elements that correspond to the algorithm type, range, reading, writing and the comparison criterion, respectively.

The concepts which are obtained from the requirement specification (F2) and through the domain analysis (F3) should be merged together. The concepts *sort*, *read* and *write* in $C_{Library}$ correspond to the concepts *AlgT*, *RD* and *WR* in C_{Sort}, respectively. We prefer the names used in the requirement specification. This results in the following set of concepts:

$$C_{Library} = (Library, Collection, LinkedList, OrderedCollection, Array, collectionItems, sort, RN, read, write, CR) \qquad (F4)$$

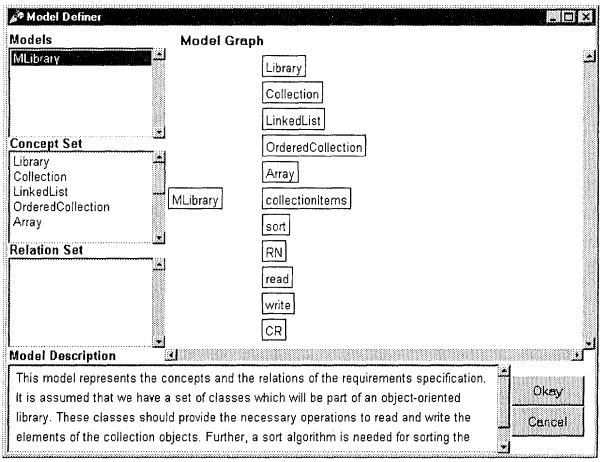


Figure 2: New models can be entered by the tool *Model Definer*

We have developed various tools to implement the techniques explained in this chapter. As shown by Figure 2, the tool *Model Definer* is used to introduce new models such as *MLibrary*. Using the tool, for each model its

concepts and their relations can be defined. Every model is stored in a global repository and can be accessed by other tools in the system.

3.3 Identification of the Adaptable Concepts

Adaptability can be defined as the ease of changing an existing model to new requirements. To this aim, we have to deal with two contradictory goals: On one hand we have to fix the concepts for robustness and time performance. On the other hand, we need to make concepts adaptable for flexibility [18]. The predefined property P_{Adapt} consists of two elements *FX* and *AD*, which is used to qualify concepts as fixed or adaptable, respectively[3]:

$$P_{Adapt} = (FX, AD) \qquad (F5)$$

Classification of concepts as fixed or adaptable creates alternative concept sets with different adaptability characteristics. The predefined properties support the techniques introduced in this chapter. These properties are implemented by the tools. We define the term *design space* to represent a set of design alternatives. Formally, design spaces are defined as function spaces that map concepts to properties. Consider, for example, the following design space:

$$S_{AdaptLibrary} :: C_{Library} \rightarrow P_{Adapt} \qquad (F6)$$

The space $S_{AdaptLibrary}$ maps the concepts of $C_{Library}$ to the elements of P_{Adapt} and as such represents the total set of alternatives of library models with adaptability properties. Every alternative can be considered as a specific design decision. For example, the following tuples may be an alternative from this space:

$$CAdaptLibrary = \{(AD, Library), (AD, Collection), (FX, LinkedList), (FX, OrderedCollection), (FX, Array), (AD, collectionItems), (AD, sort), (FX, RN), (AD, read), (AD, write), (AD, CR) \qquad (F7)$$

This alternative defines the design decision in which the range *Library*, *Collection*, *collectionItems*, *sort*, *read*, *write* and *CR* concepts have been selected as adaptable (AD) and the other concepts as fixed (FX). There are many more alternatives in $S_{AdaptLibrary}$. The total number of alternatives can be computed using the following formula:

[3] Note that other adaptability models can be identified as well. For example, at this stage of design we may also distinguish between compile-time and run-time adaptability. We are currently experimenting with different adaptability models [17].

$$totalNoAlternatives(S_{AdaptLibrary}) = size(P_{Adapt})^{size(C_{Library})} \quad (F8)$$
$$= 2^{11} = 2048$$

Hereby, the function *size* returns the number of concepts of each model. The function *totalNoAlternatives* computes the number of alternatives of the design space. Obviously, the software engineer may not be interested in all the alternatives and in addition not all of them may be possible due to various constraints. The design space can be reduced by either selecting a sub-space or eliminating the set of alternatives that are not considered feasible from the perspective of the client requirements or the internal constraints. Formally, both approaches can be specified as follows:

$$S_{AdaptLibrary} :: \{C_{Library} \rightarrow P_{Adapt} \mid (cond)\} \quad (F9)$$

Here, *cond* represents the condition for reducing the design space. The condition *cond* may consist of logical connectives to specify complex conditions. Assume, for example that the software engineer is only interested in the set of alternatives in which the concepts range (RN), read (RD), write (WR) and comparison (CR) are adaptable. This can then be formally expressed as follows:

$$S_{AdaptLibrary} :: \{C_{Library} \rightarrow P_{Adapt} \mid (RN \rightarrow AD) \wedge (RD \rightarrow AD) \wedge (WR \rightarrow AD) \wedge (CR \rightarrow AD)\} \quad (F10)$$

This reduces the number of the total set of feasible alternatives to $2^7 = 128$. Other selection and/or elimination conditions may be easily specified to further reduce the set of alternatives. The elimination conditions can be defined in the same manner by using a negation connective before the specified condition.

We can quantify the design alternatives to order and compare these with respect to different criteria. To reason about the adaptability of each alternative in the acquired set we assign a natural number to each model. This can be specified as follows:

$$S_{AdaptLibrary} :: \{C_{Library} \rightarrow P_{Adapt}\} \rightarrow N \quad (F11)$$

The basic goal of this quantification of alternatives is that it helps to explicitly reason about each alternative with respect to the corresponding

quality factor. The adaptability degree for each alternative may depend on various conditions[4].

In the following, we will describe the tools that implement the above operations for composing design spaces, quantifying design alternatives and generating design alternatives.

Figure 3 represents a snapshot of the tool *Design Space Composer* that can be used to define and depict the concept spaces. The software engineer can select a model and a property-set, and likewise can compose different design spaces. In Figure 3, the design space *AdaptLibrary* is composed from the property set *Adapt* and the model *MLibrary.* As it is displayed in the tool, *AdaptLibrary* includes 22 tuples, which have been generated by taking the Cartesian product of the sets *Adapt* and *Library.* The tool provides also means to use other set manipulation operations to refine design spaces. In this chapter, however, we will not elaborate on these and refer for a more detailed description to [17].

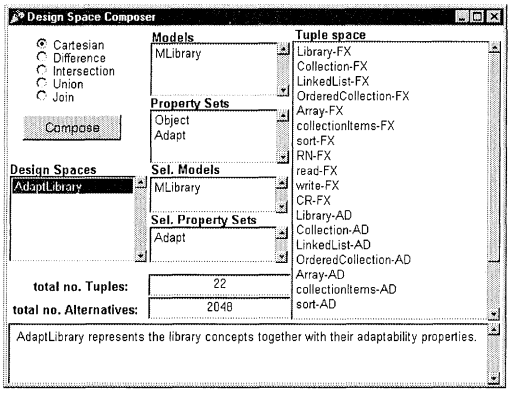


Figure 3: A tool for composing design spaces

Figure 4 represents a tool for quantifying the tuples of a design space. The top-right widget *Priority* displays the priority number for each generated tuple. Depending on the concept type, different priorities can be assigned to the tuples. For example, if adaptability is considered important, then a higher priority value can be given to the adaptable tuples. In Figure 4, among the adaptable tuples, the concept *sort* has the highest priority and range *RN* has the least. The tuples *read* and *write* have the same priority. The software engineer here assumes that from the perspective of adaptability, changing the

[4] This value should be interpreted as an indication of the software engineer's preference. The assigned adaptability degree does not imply that it is also available. During the further refinement process when other concerns are introduced it may appear that the required adaptability is not possible due to for example language constraints.

algorithm type has the most impact and changing the range the least. When a priority value is changed, the tool automatically computes the new values. This allows the software engineer to experiment with the priority values.

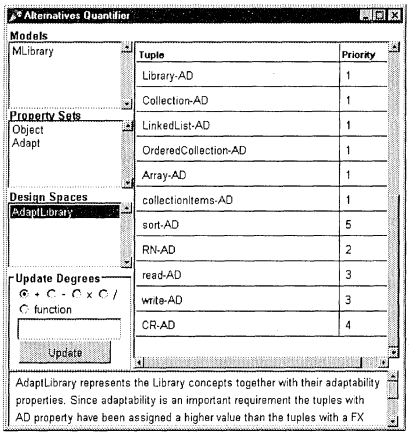


Figure 4: A tool for quantifying design alternatives

To generate alternatives from the predefined design spaces the tool *Alternative Generator* is used from which a snapshot is shown in Figure 5. Initially, the set of alternatives for the design spaces listed in the list box *Design Spaces* is not generated. The widget *no. alternatives* defines the number of alternatives that can be derived from the selected design space. The software engineer can generate the set of alternatives by pressing the *Generate* button. Since this number of alternatives can be quite large, the tool gives an error message when the number of alternatives exceeds a predefined maximum value. If the number of alternatives is smaller than the maximum default value the alternatives will be generated and listed and ordered according to their priority values. This ordering of the alternatives is also shown in the graphic below the list of alternatives. In the graphic each point represents an alternative. The graphic shows only 30 alternatives at once. To browse the other alternatives the left and right arrows at the right corner of the window can be used.

The software engineer can directly select some of these alternatives through the menu of the *alternatives list* and store this in the repository. The design space can also be reduced by either pressing the button *Matrix Selection* or the button *Rule-Based Selection* that opens tools for conditional selection of the sub-spaces and heuristic rule supported selection, respectively.

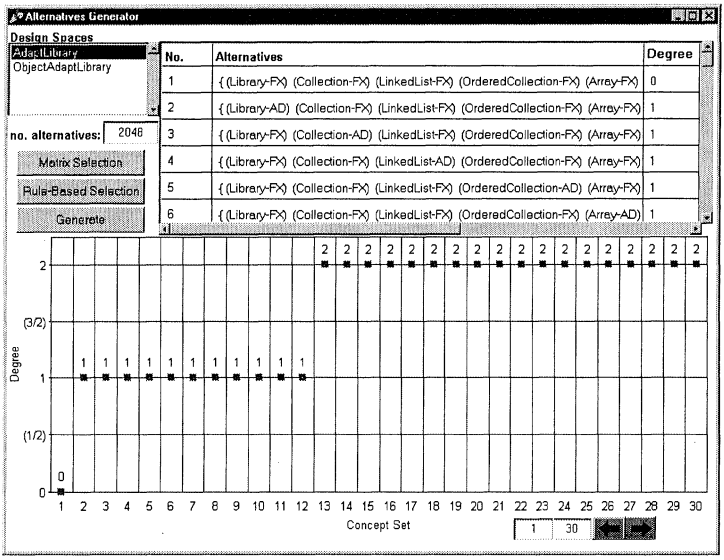


Figure 5: The tool *Alternatives Generator* is used to generate concept sets from concept spaces

Figure 6 shows the dialog window that is opened if the software engineer presses the button *Rule-Based Selection.* The window shows a dialog with questions that the software engineer needs to select the relevant tuples. In the collection library example, 11 questions are asked. As a reply, the software engineer may choose *Yes*, *No* or *I don't know*. In case *I don't know* is selected, all the alternatives for the corresponding concept are kept. The number of the sets to be generated is displayed in the dialog as well.

The reduced design spaces can be stored as a new design space and imported by the relevant tools for further refinement. After selecting the tuples, and generating the alternatives the software engineer may then compare these based on their adaptability degrees.

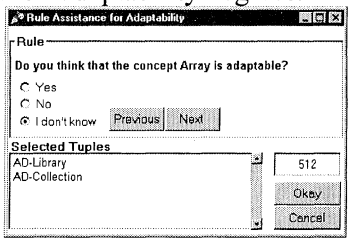


Figure 6: The dialogue that helps the software engineer to select the required adaptable tuples

3.4 Identification of the Object-Oriented Abstractions

In the object-oriented model, a concept can be represented either as a class, an operation or an attribute. The predefined property P_{Object} is a set of object modeling alternatives for concepts:

$$P_{Object} = (CL, OP, AT) \quad (F12)$$

Here, *CL*, *OP* and *AT*, represent classes, operations and attributes, respectively. We can further classify operations as virtual (Op_v) and not virtual (Op_n) operations, attributes as mutable (AT_m) and constant attributes (At_c), etc[5]. Virtual operations can be polymorphically overridden through inheritance. Constant attributes cannot change at run-time.

The following concept space $S_{ObjectAdaptLibrary}$ maps the concepts of $C_{AdaptLibrary}$ to the elements of P_{Object} and as such represents the total set of alternatives of library models with adaptability properties:

$$S_{ObjectAdaptLibrary} :: C_{AdaptLibrary} \rightarrow P_{Object} \quad (F13)$$

Again, this design space may be reduced by selection functions:

$$S_{ObjectAdaptLibrary} :: \rightarrow \{C_{AdaptLibrary} \rightarrow P_{Object} \mid (cond)\} \quad (F14)$$

We have to extend the adaptable concept model with the property *objectModel* to store one of the selected object-oriented abstractions, which is either *CL*, Op_v, Op_n, AT_m or AT_c.

The tool *Alternatives Generator*, which was shown in Figure 4 for selecting the adaptable library concepts, can be used here as well. This space is generated from the property *Object* and the concept set *AdaptLibrary*. The software engineer may directly select alternatives from this space or first reduce it. When the button *Rule Assistance* is pressed, a dialog window is opened[6] such as shown in Figure 7. During a dialog session, the following questions could be asked to the software engineer:

IF THE TUPLE IS ADAPTABLE:
(R1) **IF** THE CONCEPT REPRESENTS AN OPERATION AND RUN-TIME ADAPTABILITY IS REQUIRED,
THEN DEFINE IT AS A PAIR OF OPERATION (OP_V) AND CLASS (CL);

[5] It is also possible to define derived classes and attributes.

[6] This dialog window is generated by the property *Object*. In this way, the tool *Alternatives Generator* remains generic. Similarly, the dialog window shown in Figure 6 was generated by the property *Adapt*.

(R2) **IF** THE CONCEPT REPRESENTS AN OPERATION AND COMPILE-TIME ADAPTABILITY IS REQUIRED,
THEN DEFINE IT AS A VIRTUAL OPERATION (OP_V);

(R3) **IF** THE CONCEPT IS AN ATTRIBUTE, **THEN** DEFINE IT AS A MUTABLE ATTRIBUTE (AT_M).

(R4) **IF** (R1) TO (R3) ARE NOT APPLICABLE **THEN** DEFINE THE CONCEPT AS A CLASS (CL);

IF THE TUPLE IS FIXED:

(R5) **IF** THE CONCEPT IS AN ATTRIBUTE,
THEN DEFINE IT AS A CONSTANT ATTRIBUTE (AT_C);

(R6) **IF** THE CONCEPT IS AN OPERATION
THEN DEFINE IT AS A (NON-VIRTUAL) OPERATION (OP_N)

(R7) **IF** (R5) AND (R6) ARE NOT APPLICABLE
THEN DEFINE THE CONCEPT AS A CLASS (CL).

The rule (R1) is assumes that if the concept is an operation and is run-time adaptable, then it must be represented as an operation declared at the interface of a mutable object (like the Strategy pattern). According to the rule (R2), if the concept is an operation and is compile-time adaptable, then it can be defined as a virtual (abstract) method (like the Template Method pattern). If, however, the concept is an attribute and it is adaptable, then it can be defined as a mutable attribute (R3). The rule (R4) assumes that if the concept is adaptable and the rules (R1) to (R3) are not applicable, then the concept can be defined as a class. The rule (R5) suggests that if the concept is a fixed attribute, then it can be defined as a constant attribute. According to the rule (R6), if the concept is a fixed operation, then it can be selected as a non-virtual operation. Finally the rule (R7) assumes that if the fixed concept is neither an attribute nor an operation, then it can be represented as a class. Note that the rules (R4) and (R7) are quite similar, since they both select the class abstraction. We could make distinction between these rules by assuming that (R4) and (R7) create mutable and constant classes (objects), respectively. In most object-oriented languages, however, objects are per default mutable. Additional programming effort is necessary to enforce constant objects. We therefore do not make distinction between mutable and constant objects.

These rules considerably simplify the generation of a concept set. Although the possible number of alternative concept sets is $5^{11} = 48828125$, by using the heuristics rules, a concept set can be selected only in 11 steps. Similar to the dialog window shown in Figure 6, if the software engineer selects the button *I don't know*, then the alternative concept sets are

generated. Again, these alternatives can be compared with respect to their adaptability degrees[7].

We will now illustrate object-oriented concept identification process using the collection classes example. Assume that based on the functions (F7) and (F14), and by using the tool *Alternatives Generator* the software engineer decides on the following adaptability properties:

$$C_{ObjectAdaptLibrary} = \textit{((CL, AD, Library), (CL, AD, Collection), (CL, FX, LinkedList), (CL, FX,OrderedCollection), (CL, FX, Array), } (AT_m, \textit{ AD, collectionItems), } (OP_v, \textit{ AD, sort), (CL, AD, sortClass), } (AT_c, \textit{ FX, RN), } (OP_v, \textit{ AD, Read), } (OP_v, \textit{ AD, write), } (OP_v, \textit{ AD, CR), (CL, AD, CRClass))} \qquad \textit{(F15)}$$

Here, the adaptable tuples *(AD, Library)*, *(AD, Collection)*, *(AD sort)* and *(AD, CR)* are considered as classes. Note that based on the rule (R1), to represent sort and CR two new tuples are introduced. To distinguish operation and class names, we use the names *sort*, *sortClass*, *CR* and *CRClass*. The adaptable tuples (AD, read) and (AD, write) are represented as virtual operations. The adaptable tuple (AD, collectionItems) is considered as a mutable attrubute. The fixed tuples (FX, LinkedList), (FX, OrderedCollection) and (FX, Array) are considered as constant classes. The fixed tuple (FX, RN) is considered as a fixed attribute.

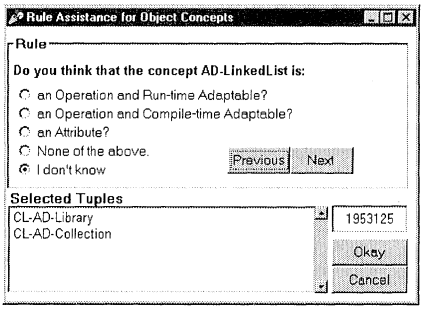


Figure 7: The dialogue used to generate object-oriented concepts

3.5 Identification of the Object-Oriented Relations

We will now consider the relations among concepts. As an example, consider the following table, which represents the relations among the

[7] This tool is implemented as an expert system, which applies object-oriented heuristics to reduce the design space. Various different heuristics can be defined by using the method engineering facilities of the tool [17].

concepts of $M_{Library}$. This table is derived from the requirement specification and domain analysis[8].

Table 1: Identification of relations among the concepts of Library

Concept	has a relation with the following concepts
Library	Collection, LinkedList, OrderedCollection, Array
Collection	LinkedList, OrderedCollection, Array
CollectionItems	Collection, LinkedList, OrderedCollection, Array
Sort	Collection, LinkedList, OrderedCollection, Array, SortClass
Sort	RN, read, write, CR
CollectionItems	RN, read, write, CR, sort

The top 4 rows are directly derived from the requirement specification. Here, it is assumed that the concept *Library* contains the collections. The second row indicates that *Collection* is an abstraction of *LinkedList*, *OrderedCollection* and *Array*. The third row indicates the relation between *collectionItems* and the collections. The fourth row represents the relation between sorting and the collections.

The rows 5 and 6 are derived from the sorting domain. The fifth row indicates the relation between the sorting algorithm and range determination, reading, writing and comparison operations. The sixth row represents the relation between *collectionItems* and range determination, reading, writing and comparison operations.

After the identification of the object-oriented abstractions (like in (F14)) additional tuples and therefore new relations may be introduced. For example, in previous section we have replaced *(AD, sort)* and *(AD, CR)* by the pair of concepts *(OP_v, AD, sort)* and *(CL, AD, sortClass)* and *(OP_v, AD, CR)* and *(CL, AD, CRClass)*. By using the tool *Model Definer* new concepts can be easily introduced at any time. The other related tools are updated automatically.

The relation among the concepts can be represented as a pair of concepts. Consider, for example the following lists of tuples which are derived from Table 1 after the adaptability and object-oriented properties of the concepts have been selected:

[8] The relations can be derived using any object-oriented analysis method.

$R_{ObjectAdaptLibrary}$ = *(((CL, AD, Library), (CL, AD, Collection)), ((CL, AD, Library), (CL, FX, LinkedList)), ((CL, AD, Library), (CL, FX, OrderedCollection)), ((CL, AD, Library), (CL, FX, Array)), ((CL, AD, Collection), (CL, FX, LinkedList)), ((CL, AD, Collection), (CL, FX, OrderedCollection)), ((CL, AD, Collection), (CL, FX, Array)), ((ATm, AD, collectionItems), (CL, AD, Collection)), ((ATm, AD, collectionItems), (CL, FX, LinkedList)), ((ATm, AD, collectionItems), (CL, FX, OrderedCollection)), ((ATm, AD, collectionItems), (CL, FX, Array)), ((OPv, AD, sort), (CL, AD, Collection)), ((OPv, AD, sort), (CL, FX, LinkedList)), ((OPv, AD, sort), (CL, FX, OrderedCollection)), ((OPv, AD, sort), (CL, FX, Array)), ((OPv, AD, sort), (ATc, FX, RN)), ((OPv, AD, sort), (OPv, AD, read)), ((OPv, AD, sort), (OPv, AD, write)), ((OPv, AD, sort), (CL, AD, CR)),(OPv, AD, sort), (CL, AD, sortClass), ((ATm, AD, collectionItems), (ATc, FX, RN)), ((ATm, AD, collectionItems), (OPv, AD, read)), ((ATm, AD, collectionItems), (OPv, AD, write)), ((ATm, AD, collectionItems), (CLm, AD, CR)), ((OPv, AD, CR), (CL, AD, CRClass)))* *(F16)*

Note that the new concepts *sortClass* and *CRClass* are added to the tuple-set as well.

The predefined property $P_{ObjectRelation}$ is a set of object modeling alternatives for relations:

$$P_{ObjectRelation} = (AG, IN, WS, CS, RS, WS) \quad (F17)$$

Here, *AG*, *IN*, *WS*, *CS*, *RS* and *WS*, represent *aggregation*, *inheritance*, *owns*, *calls*, *reads* and *writes* relations. These relations are explained in Table 2.

Table 2: Object-oriented relations

FROM \ TO	A. *CL*	B. OP_v	C. OP_n	D. AT_m	E. AT_c
1. *CL*	aggregates/ inherits from	owns	owns	owns	owns
2. OP_v	owned by/ inherited from	calls (direct/inherited)	calls (direct/inherited) /in-lines	reads -writes	reads
3. OP_n	owned by	calls (direct/inherited)	calls (direct/inherited) /in-lines	reads -writes	reads
4. AT_m	owned by/ inherited from	read-written by	read-written by	derived from	derived from
5. AT_c	owned by	read by	read by	derived from	derived from

In this table, the columns and rows correspond to the object-oriented constructs in a relation set. The elements of the table represent the possible

object-oriented relations. Since most object-oriented relations are directed, we define the direction of the relations from the concepts of the first column to the concepts of the first row. For example, the relation 2D should read as OP_v reads-writes AT_m.

The relation 1A indicates that the possible relations between two classes are aggregate and inheritance relations. The relations 1B to 1E are all *owns* relations because operations and attributes must always belong to a class. In the second row, the relations 2B and 2C specify that OP_v can call on OP_v or OP_n. Sometimes, the calls on fixed operations can be in-lined in the implementation of the calling operation. Similarly, relations 3D and 3E mean that OP_v can read and write on AT_m but can only read from AT_c. The other relations are self-explanatory.

The following relation space $S_{ObjectRelationAdaptLibrary}$ maps the relations of $R_{AdaptLibrary}$ to the elements of $P_{ObjectRelation}$ and as such represents the total set of alternatives of the object-oriented relations in the adaptable library:

$$S_{ObjectRelationAdaptLibrary} :: \rightarrow \{ R_{AdaptLibrary} \rightarrow P_{ObjectRelation} \mid (cond) \} \qquad (F18)$$

Here, *cond* represents the restrictions on the possible relations. The restrictions must be derived from the semantics of the application, as defined in Table 1. In addition, as shown in Table 2, in the object-oriented model only a certain kinds of relations are possible.

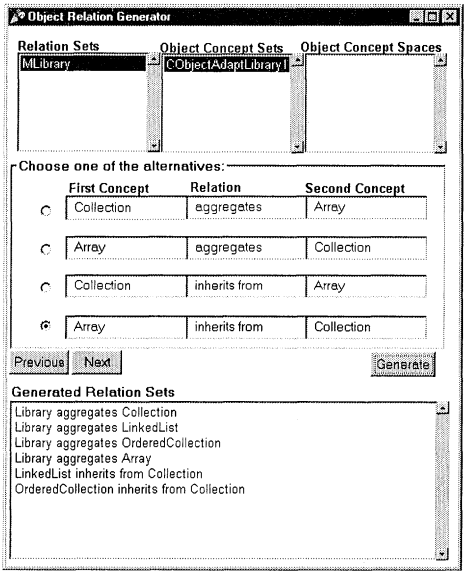


Figure 8: A tool for generating object-oriented relations

The tool *Object Relation Generator* as shown in Figure 8 helps the software engineer to select the appropriate relation. The top widgets *Relation Sets* and *Object Concept Sets* list the relations and concept sets of the models. The selected concept sets must have their P_{Object} values defined. It is also possible to consider alternative concept sets by selecting an item from the widget *Object Concept Spaces*.

Consider the following example. According to the relation 1.A from Table 2, the first relation ((CL, AD, Library), (CL, AD, Collection)) may have 4 possible implementations: These are "Library aggregates Collection", "Collection Aggregates Library", "Library inherits from Collection" and "Collection inherits from Library". We assume that the software engineer selects the "Library aggregates Collection" relation. If the relations were defined as directed relations, then the tool would only propose 2 object-oriented relations. The tool iterates through all the relations.

Now assume that based on the selections of the software engineer, the following relation set is generated:

$R_{\text{ObjectRelationAdaptLibrary}}$ = *(Library aggregates Collection), (Library aggregates LinkedList), (Library aggregates OrderedCollection), (Library aggregates Array), (LinkedList inherits from Collection), (OrderedCollection inherits from Collection), (Array inherits from Collection), (collectionItems owned by Collection), (collectionItems owned by LinkedList), (collectionItems owned by OrderedCollection), (collectionItems owned by Array), (Collection owns sort), (LinkedList owns sort), (OrderedCollection owns sort), (Array owns sort), (sortClass owns sort), (sort reads RN)), (sort calls read), (sort calls write), (sort calls CR), (RN reads collectionItems), (read reads collectionItems), (write writes collectionItems), (CR reads-writes collectionItems), (CRClass owns CR)* (F19)

The relations in (F19) can be largely simplified by using object re-factoring rules [11]. For example, common aggregation relations, attributes and methods can be moved to their super class, if any. In (F19) the aggregation relations between *Library* and *Collection*, *LinkedList*, *OrderedCollection* and *Array* can be reduced to "*Library aggregates Collection*" because all other classes inherit from *Collection*. Similarly, the aggregation relations between the collection classes and *sort* can be reduced to "*Collection aggregates sort*". The "owned by" relations between the *collectionItems* and the collection classes can be reduced to "*collectionItems owned by Collection*". Further if an operation is owned by multiple classes and if there is an inheritance relation between these classes, then the

operation can be moved to the super class. If these classes do not inherit from each other, then the method can be replicated in every class. The re-factoring of the object diagram can be advised or largely realized by a tool. In the appendix, the class diagrams of the six selected alternative implementations of *MLibrary* after the re-factoring process are shown.

4. DESIGNING FOR TIME PERFORMANCE

This section introduces a simple process to determine the time performance factors of models at various abstraction levels. These factors can be used to compare the alternative models. The performance analysis process is based on simulation. For this purpose, we have developed a simulation environment and a set of tools. This environment involves a set of random generators, time measurement and display units[9]. The performance analysis process involves the following steps:

1. Construction of the model: To determine the probabilistic time performance-value of a model, first its concepts and relations must be identified. It is possible to determine the relative time performance of a model even before its adaptability and object properties are determined.
2. Identification of the behavioral concepts: We need to identify the concepts that contribute to the behavior of the model. For example in *MLibrary*, the concept *sort* is the most significant concept for the sorting process.
3. Identification of the interaction diagram: An interaction diagram specifies a call pattern among the related concepts. For example, when the operation *sort* is invoked, depending on the sorting algorithm, *sort* calls on other concepts to realize the sorting process. To simulate a model it is necessary to define a sub-graph, which represents the interaction diagram of that model. For example, the sub-graph shown in Figure 9 is the interaction diagram of *MLibrary* for sorting the items in the collection objects. For simplicity, here the direction of calls is not shown.

[9] The emphasis of this section is not to introduce a new performance analysis technique but adopt an existing technique to compare various design alternatives. In the literature, many performance analysis techniques have been published [15][8].

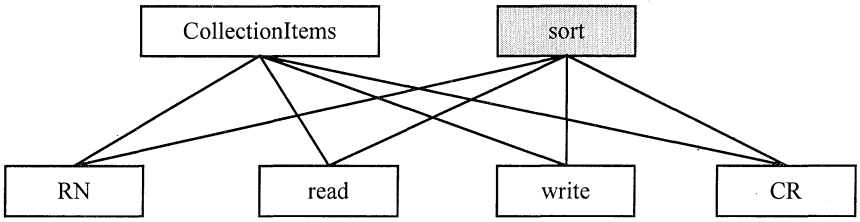


Figure 9: The interaction of concepts in *MLibrary* for sorting the items

4. Specification of the behavior: The concepts identified in step 2 must be specified. For example, for *MLibrary*, we have searched for various sorting algorithms in the sorting domain [14]. We have then selected the *bubble* and *selection* sort algorithms as the two possible implementations of the sorting process.
5. Implementation of the simulation graph: We have developed a framework to implement the simulation graphs such as the one shown in Figure 9. The nodes of this graph must be specialized within the simulation context. For *MLibrary*, we created 6 nodes and then specialized these nodes according to the graph shown in Figure 9[10].
6. Determination of the simulation parameters: This involves parameterization of the random generators, the range of simulation, etc. For example, we have simulated the interaction diagram of *MLibrary* in 2 different environments. In the first setting, we fixed the number of items in the collection objects to 100, we run the sorting process 100 times, and we randomly generated the items in the collections using the *Linear Congruential Generator* algorithm [12]. In the second setting we changed the number of items in the collections randomly.
7. Determination of the probability values of calls: Using the facilities of the simulation environment, the number of calls per relation must be counted and normalized with respect to the total number of calls. These values represent the probability of calls. Figure 10 shows the tool, which displays the probability values of calls of *MLibrary*. Here, figures 10(a) and 10(b) display the simulation environment for the first and second settings, respectively[11].

[10] Depending on the application, various simulation languages can be used [8][15].
[11] The purpose of this phase is to obtain reliable probability values. The detailed analysis of this topic is considered beyond the scope of this chapter.

Note that the probability of calls to RN (range calculation) is very small and therefore can be neglected. The relations between classes do not directly influence the performance and therefore they are set to zero. We can now determine the relative performance values of alternative implementations of *MLibrary*, provided that the interaction patterns remain the same. As illustrated in the previous sections, a model can be implemented in many different ways, and the relative performance factors of models will depend on the type of the object-oriented relations used[12]. We would like to reason about the alternatives by assigning a time performance value to each model.

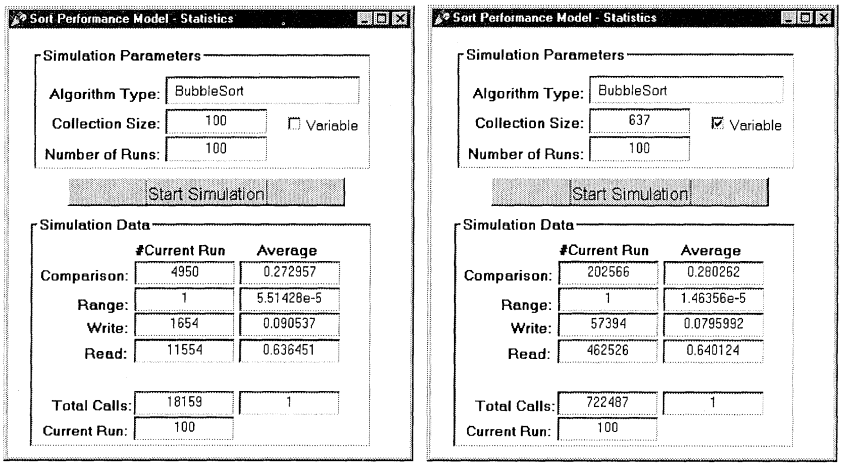


Figure 10: The frequency of calls in *MLibrary*: (a) with 100 items, (b) with randomly varying number of items.

The relative time performance value of a model is computed in the following way:

$$performance = 100 / \Sigma\, p_i\, r_i \quad (F21)$$

Here, i is the index of the set, where p_i and r_i represent the probability and relative cost values of the indexed element of the set, respectively. The cost calculation must be iterated through the range of the set. We multiply the result with 100 for the scaling purpose. The relative cost values of relations are language dependent. Since our experimental environment is implemented in the Smalltalk language, the following average cost values are measured from our Smalltalk system. These values are normalized with respect to the cost of an in-lined call:

[12] The concepts of a model influence the call relations indirectly, because they determine the type of relations used (Table 2).

Table 3: The relative cost values in the Smalltalk system

Relation type	Cost
message call	5
inherited call	4
inline	1
attribute read	5
attribute write	16

In the previous sections, the probability of calls was assumed to be the same for every alternative model. This assumption is valid if the behavioral concepts of the models remain the same. If these concepts are adaptable, then we need to build a new simulation model for each interaction pattern. For example, in addition to the *bubble sort* algorithm, we also simulated the *selection sort* algorithm. As published in the literature [14], the *selection sort* algorithm generally performs better than the *bubble sort* algorithm. However, if the items in the collection are almost sorted, then the *bubble sort* is faster. Therefore the choice of an algorithm depends on the context of execution. A more detailed discussion about simulating multiple alternative algorithms is given in [17].

5. BALANCING ADAPTABILITY AND PERFORMANCE FACTORS

The appendix of this chapter shows the adaptability and performance values of the six design alternatives of the collection library. We will now compare these alternatives. In Figure 11 the Y-axis and X-axis show the performance and adaptability degrees of the alternative models, respectively. The models are represented as diamonds in the graph. The model numbers are shown at the right of the corresponding diamonds. We would like to emphasize that these models cannot be compared by considering their relative performance and preferred adaptability degrees only. The definition of these models as shown in the appendix must be considered as well.

Model 1 has the highest performance and lowest preferred adaptability value. This is of course an expected result since the implementation of the sort algorithm is in-lined. For the total scale we can observe that a higher preferred adaptability degree eventually results in a lower performance degree. This conclusion, however, is not necessarily valid for models with close preferred adaptability degrees. As it can be derived from Figure 11, Model 2 provides a slightly higher preferred adaptability degree than model

4 but still has a higher performance degree. In Model 2 the comparison criterion is the only adaptable operation whereas the other operations are in-lined. In model 4 two operations are compile-time adaptable, that is read and write, and the comparison operation is in-lined. Thus, in model 4 more components are adaptable than in Model 2. This results in a lower performance for model 4. From this we can conclude the following: To fulfil the flexibility requirements, sometimes a concept must be made run-time adaptable. In this case, it may be still possible to obtain a high performance, if all other concepts are fixed.

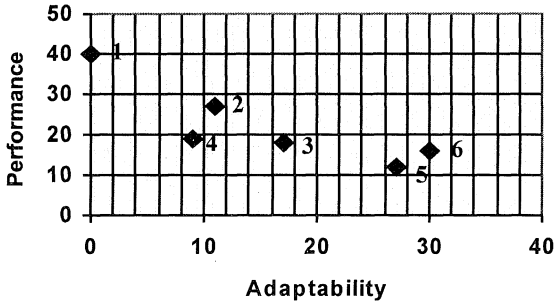


Figure11: The relative cost values of the selected design alternatives

Model 3 is similar to Model 4. In Model 4, the comparison criterion was made run-time adaptable but to compensate the performance loss, all other operations were fixed. In Model 3, however, no such compromise is made. The performance difference between Model 3 and Model 4 shows the penalty paid for making the comparison criterion run-time adaptable without making any compromise.

Model 6 and Model 5 have both four operations adaptable. In Model 6 all the four selected operations are run-time adaptable whereas in Model 5 two of the four operations are compile-time adaptable. Although Model 6 has thus a higher preferred adaptability degree than Model 5 it performs slightly less. The main reason for this is that in Model 5 the operation *sort* is made adaptable whereas in Model 6 this is made fixed. Since this operation has the highest adaptability priority (5) it substantially increases the preferred adaptability degree of model 5. Nevertheless, this operation has a minor effect on the performance degree. This means that if an operation is not executed frequently, making that operation adaptable will cause practically no performance degradation. We can also conclude that run-time adaptability increases the preferred adaptability degree but has a minor additional effect on the performance degree with respect to compile-time adaptability.

The presented results are computed for the *bubble* sort algorithm and the distribution of the collection items to be sorted is the same for all the models. The *bubble* sort algorithm, however, is faster than the *selection* sort algorithm if the items in the collection objects are largely sorted. Model 5 allows run-time substitution of the sorting algorithm and the comparison criterion. The performance degree of this model is 12. If the number of the items to be sorted continuously changes, then this model may have a better average time performance[13]. Obviously, if the probabilistic behavior of the change is known, it is possible to compute when implementations that allow behavioral change provide a better time performance.

To verify the validity of the relative time performance values of the selected models, we have implemented the models 1, 2 and 6. We have generated the collection items using the same simulation environment as shown in Figure 10. The time performance values of the models 1, 2 and 6 were measured as 585, 910 and 1310 milliseconds, respectively. In the following, the ratio of the measured values is compared with the estimated values. We can conclude that the estimated values are reasonably accurate.

measured	1310/585 = 2.2	1310/910 = 1.4	910/585 = 1.6
computed	40/16 = 2.5	40/27 = 1.5	27/16=1.7

From Figure 11, the software engineer can select on the Y-axis an acceptable time performance, say 18, and can determine the affordable degree of preferred adaptability from the X-axis. Note that by using the table above, it is also possible to convert the relative values to the actual time performance factors.

6. RELATED WORK

Adaptability is generally considered as an important and desired characteristic of software systems and a number of research groups have been active in this area. For example, to improve the adaptability characteristics of software systems, the Demeter method [34], Composition-Filters [1], Aspect-Oriented Programming [7], and Reuse Contracts [16] are proposed as extensions to the object-oriented model. We consider these contributions important and complementary to our work. Our emphasis, however, is different. We do not propose an extension the to object-oriented

[13] Part of the performance penalty in this model is due to run-time adaptability of the comparison criterion.

model, but introduce a technique to compare the design alternatives from adaptability and performance viewpoints.

In [6], the concept of variation point is introduced to specify locations at which variation will occur. The variation points are generally expressed using variants, which are type-like constructs. Although our adaptability modeling approach is intuitively similar, we propose an adaptability model, which can be applied along the software development process for comparing the design alternatives.

Several publications have been made on object-oriented software metrics [4]. Software metrics is quantitative measurements about any aspect of a software project. This may include *project, process and product metrics.* Product metrics aim to determine the properties of the software product, such as the amount of coupling, cohesion, code complexity, etc. Most product metrics as published in the literature are generally determined after the software system is built and there is no clear relation between the quality demands of requirements, compromises being made, and the quality of systems being built.

Simulation techniques have been used in analyzing the performance of software systems for many years [8]. In our example method, we applied a simple simulation technique to determine the relative performance characteristics of the design alternatives. In this chapter our intention is not to introduce a new sophisticated performance analysis method, but rather adopt an existing suitable technique.

During the last decade, the so-called Software Performance Engineering (SPE) discipline has emerged for combining the performance analysis techniques with software engineering methods [15]. This discipline aim to construct performance models of software systems by using data about envisioned software processing. These models are used to compare software and hardware alternatives for solving performance problems. The techniques used within the context of SPE research are relevant to our work, and can be applied together with the techniques presented in this chapter. Our emphasis is to compare the design alternatives both from performance and adaptability viewpoints, whereas the SPE research mainly emphasized the performance factors of the design alternatives.

The techniques presented in this chapter can be considered as a special form of Relational Algebra [5]. Our tools implement operations that are similar to the *union*, *product*, *select* and *join* operations of Relational Algebra. The select operation in our case is based on design heuristics. We therefore term our technique as Design Algebra. We are currently applying and formalizing Design Algebra within the context of a large transaction system design [17].

Generally speaking, our work can be classified under the so-called "AI-based problem solving techniques" [9][19]. These techniques generally implement a problem solution strategy and a set of heuristics to guide the engineers in implementing their designs. Most of the work in this area, however, is in designing mechanical or electronic systems.

7. EVALUATION

For several years, we have been applying Design Algebra and the related tools in various projects [17]. Further, we tutored Design Algebra in various conference tutorials[14] and professional courses. Based on these experiences, we will now evaluate Design Algebra from the perspective of scalability, complexity of the design space, complexity of the process, adaptability and performance analysis.

Scalability: In this chapter we use a rather simple example for illustrative purposes. In practice, however, we applied Design Algebra and the related tools in larger projects such as atomic transaction system design [17]. Currently we are experimenting with Design Algebra in designing quality-aware middleware systems. With respect to scalability, we observe that the techniques presented in this chapter do not necessarily increase the complexity of current object-oriented practices. For example, the only required extensions to the UML models are the introduction of additional attributes for storing the adaptability properties and preferred adaptability values. From methodological point of view, in addition to applying familiar object-oriented design rules, in our case the software engineers have to determine whether a concept should be adaptable or fixed. If desired, the preferred adaptability values for concepts may be defined as well. This does not necessarily complicate the process, since only a few questions have to be answered. In addition, this process is supported by the tools. The performance analysis process is limited to a comparative performance analysis and therefore does not require detailed performance analysis models.

Complexity of the design space: The design space concepts as formulated in (F6), (F8), (F13) and (F18) are purely conceptual. The software engineers do not deal with the total number of design alternatives. In practice, only a relevant number of alternatives, say up to 15, are

[14] The techniques presented in this chapter were partially tutored in ECOOP'98, OOPSLA'99 and ECOOP'2000 conference tutorials.

considered at a time. One may claim that even a limited number of alternatives increase the complexity of design. We think that in any realistic project alternative models are always defined. In general, these models are not managed properly and kept in various files and/or repositories. Our tools keep tract of the differences between the alternative models and provide means to compare, select and if necessary eliminate them. We think that our approach may reduce the complexity in dealing with alternatives, which may be hidden in the project environment.

Complexity of the process: The refinement process as presented in this chapter, such as problem understanding, domain analysis, object modeling and implementation is not fundamentally different from the advises of most object-oriented methods. The only additional work is to consider the adaptability values of concepts. For performance analysis we also consider the probability values of interactions.

The software engineer may introduce new concepts at any stage of the process. The tools keep tract of the changes. For example, if a new modeling element is introduced in later stage, the tool asks the software engineer if its adaptability characteristics have to be considered. We consider rule-base support as shown in figures 6 and 7 extremely useful during the design process. Further, while gathering the input for rules, the system creates an automatic documentation of the design decisions.

Adaptability analysis: The main purpose of the preferred adaptability values is to express the wishes of the software engineer, to label the alternative models and/or to order the models, if appropriate. The software engineer has always an access to the definition of the priority values and the corresponding object models. The software engineer may select various schemes for calculating the adaptability value of a model such a summing up the values of tuples, giving a weighting value per model, etc. What is more important is the explicit and relative consideration of the adaptability factors of concepts and their effect to the overall model. If the software engineer cannot decide on the adaptability of a concept, he/she may leave it undefined.

Performance analysis: In this chapter we have presented a simple technique to analyze the relative performance of the alternatives. There are a great number of successful techniques for performance analysis and the software engineer should adopt the appropriate one. However, relative performance analysis is generally simpler than a detailed analysis of the system, since only the differences among the models have to be considered.

8. CONCLUSIONS

There are, in general, many correct implementations of a software design problem, and each implementation may differ from the other with respect to its quality factors. Software is rarely designed for ultimate quality, but it is a compromise of multiple considerations. For example, generally the adaptability and performance factors of a software system have to be balanced. To achieve these objectives, in this chapter the following four requirements were considered important: First, to be able to compare the design alternatives, the space of the alternatives must be determined. Secondly, the alternatives must be ordered with respect to their quality factors. Thirdly, the software engineers must be able to select among the alternatives based on the requirements. Finally, the quality factors must be balanced with respect to each other.

In section 3, a process has been presented to explicitly reason about the adaptability factors of the design alternatives at various abstraction levels. In section 4, a simulation technique was used to determine the relative time performance factors of the design alternatives. To this aim, it was found sufficient to build a single simulation model. In section 5, we have analyzed six design alternatives from their adaptability and performance viewpoints. We have shown that the adaptability and time performance factors of the software systems can be balanced with respect to the requirements. Comparing these quality factors was also educational for us and our findings were summarized in section 5.

The techniques presented in this chapter can be considered as a special form of Relational Algebra, which we termed as Design Algebra. We think that the algebraic techniques provide a formal foundation and enable implementation of suitable tools. We also think that the proposed technique is practical since it can be easily integrated with current object-oriented methods. Rule-based heuristics are particularly useful in selecting and evaluating the alternatives. The algebraic techniques can be extended to other quality factors as well. We are currently working on reuse factors of design alternatives.

ACKNOWLEDGEMENTS

This research has been supported by various organizations including Siemens-Nixdorf Software Center, the Dutch Ministry of Economical affairs (SENTER), the Dutch Organization for Scientific Research (NWO, 'Inconsistency management in the requirements analysis phase' project), the AMIDST project, and by the IST Project 1999-14191 EASYCOMP.

9. REFERENCES

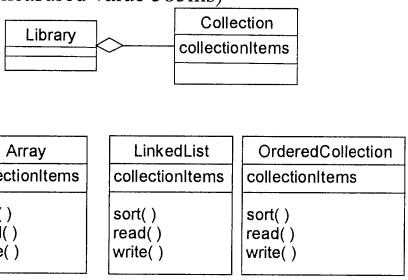
1. M. Akşit, *Separation and Composition of Concerns*. ACM Computing Surveys 28A(4), December, 1996.
2. M. Akşit, B. Tekinerdoğan, F. Marcelloni, & L. Bergmans. *Deriving Object-Oriented Frameworks from Domain Knowledge*. In Building Application Frameworks: Object-Oriented Foundations of Framework Design, Fayad et al. (eds), Wiley, 2000.
3. G. Arrango. *Domain Analysis Methods*. In Software Reusability, Schäfer, R. Prieto-Díaz, and M. Matsumoto (Eds.), Ellis Horwood, New York, New York, pp. 17-49, 1994.
4. S.R. Chidamber, & C. F. Kemerer. *A Metrics Suite for Object-Oriented Design*. IEEE Transactions on Software Engineering 20(6): 476-93, 1994.
5. C. Date. *An Introduction to Database Systems*. Addison-Wesley, 1986.
6. I. Jacobson et al. *Software Reuse*. ACM Press, NewYork, 1997.
7. G.Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier & J. Irwin, *Aspect-Oriented Programming*. ECOOP '97 Conference Proceedings, LNCS 1241, , pp. 220-242, Springer-Verlag, 1997.
8. A. M. Law & W. D. Kelton. *Simulation Modeling & Analysis*. Second Edition, McGraw-Hill, Inc., 1991.
9. C-L. Lee, G. Iyengar & S. Kota. *Automated Configuration Design of Hydraulic Systems*. In: Artificial Intelligence in Design'92, (Ed) J. S. Gero, pp. 61-82, Kluwer Academic Publishers, 1992.
10. K.J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996.
11. W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. University of Illinois, Urbana Champaign, 1992.
12. W.H. Press, B.P. Flannery, S.A. Teukolsky, W.T. Vetterling. *Numerical Recipes*. Cambridge University Press 1986, pp. 191-199, 1986.
13. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, & W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.
14. R. Sedgewick. *Algorithms*. Addison-Wesley, 1988.
15. C. U. Smith. *Performance Engineering of Software Systems*. Addison-Wesley, 1990.
16. P. Steyaert, C. Lucas, K. Mens, & T. D'Hondt. *Reuse Contracts: Managing the Evolution of Reusable Assets*. OOPSLA '96 Proceedings, ACM SIGPLAN Notices, pages 268-285, ACM Press, 1996.
17. B. Tekinerdoğan. *Synthesis-Based Software Architecture Design*. PhD Thesis, Dept. of Computer Science, University of Twente, March 23, 2000.
18. B.Tekinerdoğan & M. Akşit. *Adaptability in object-oriented software development: Workshop report*, In M. Muhlhauser (ed),Special issues in Object-Oriented Programming, Dpunkt, Heidelberg, 1997.
19. C. Tong & D. Sriram. *Introduction*. In: Artificial Intelligence in Engineering Design, Vol. 1, (Eds) C. Tong & D. Sriram, pp. 1-53, Academic Press, 1992.

APPENDIX MODELS

MODEL 1
All tuples are fixed; no practical use of inheritance.
Tuple:

$C_{ObjectAdaptLibrary}$ = *((CL, FX, Library), (CL, FX, Collection), (CL, FX, LinkedList), (CL, FX, OrderedCollection), (CL, FX, Array), (AT_m, FX, collectionItems), (OP_n, FX, sort), (AT_c, FX, RN), (OP_m, FX, Read), (OP_n, FX, Write), (OP_n, FX, CR))*

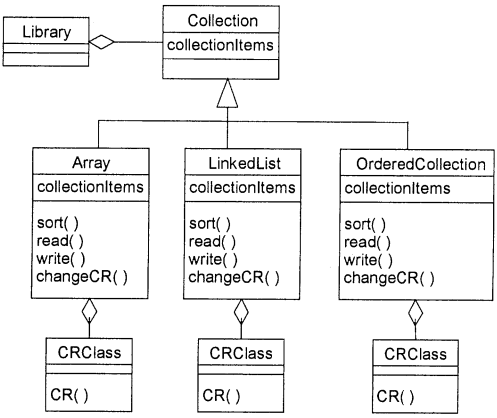
Adaptability degree: 0
Performance Degree: 40 (measured value 585ms)



MODEL 2
CR run-time adaptable, operations are non-virtual.
Tuple:

CobjectAdaptLibrary = ((CL, AD, Library), (CL, AD, Collection), (CL, FX, LinkedList), (CL, FX, OrderedCollection), (CL, FX, Array), (ATm, AD, collectionItems), (OPn, FX, sort), (ATc, FX, RN), (OPn, FX, Read), (OPn, FX, Write), (OPv, AD, CR1), (CL, AD, CRClass))

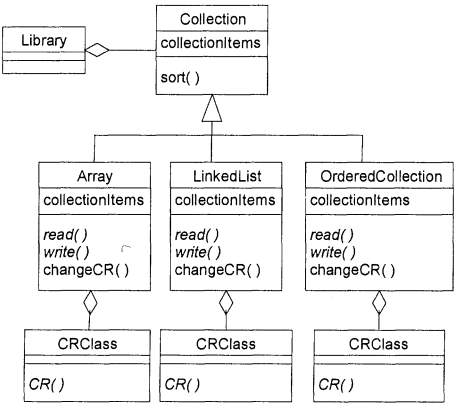
Adaptability Degree: 11
Performance Degree: 27 (measured value 910ms)



MODEL 3
CR run-time adaptable, sort is fixed but inherited.
Tuple:

$C_{ObjectAdaptLibrary}$ = *((CL, AD, Library), (CL, AD, Collection), (CL, FX, LinkedList), (CL, FX, OrderedCollection),(CL, FX, Array), (AT_m, AD, collectionItems), (OP_n, FX, sort), (AT_c, FX, RN), (OP_v , AD, Read), (OP_v , AD, Write), (OP_v, AD, CR), (CL, AD, CRClass))*

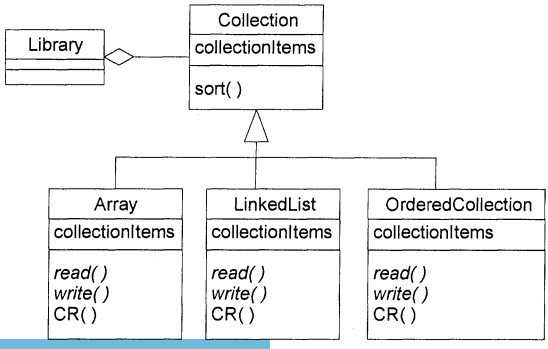
Adaptability Degree: 17
Performance Degree: 18



MODEL 4
Sort is fixed but inherited. All other methods are virtual.
Tuple:

$C_{ObjectAdaptLibrary}$ = *((CL, AD, Library), (CL, AD, Collection), (CL, FX, LinkedList), (CL, FX, OrderedCollection), (CL, FX, Array), (AT_m, AD, collectionItems), (OP_n, FX, sort), (AT_c, FX, RN), (OP_v, AD, Read), (OP_v, AD, Write), (OP_n, FX, CR))*

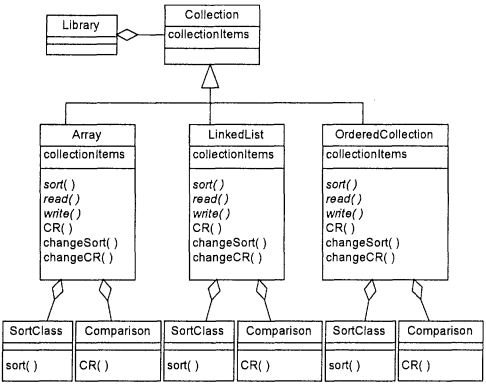
Adaptability Degree: 9
Performance Degree: 19



MODEL 5
Sort and CR run-time adaptable. All others methods are virtual
Tuple:.

$C_{ObjectAdaptLibrary}$ = *((CL, AD, Library), (CL, AD, Collection), (CL, FX, LinkedList), (CL, FX, OrderedCollection), (CL, FX, Array), (AT_m, AD, collectionItems), (OP_v, AD, sort),(CL, AD, sortClass), (AT_c, FX, RN), (OP_v,,AD, Read), (OP_v,, AD, Write), (OP_v, AD, CR), (CL, AD, CRClass))*

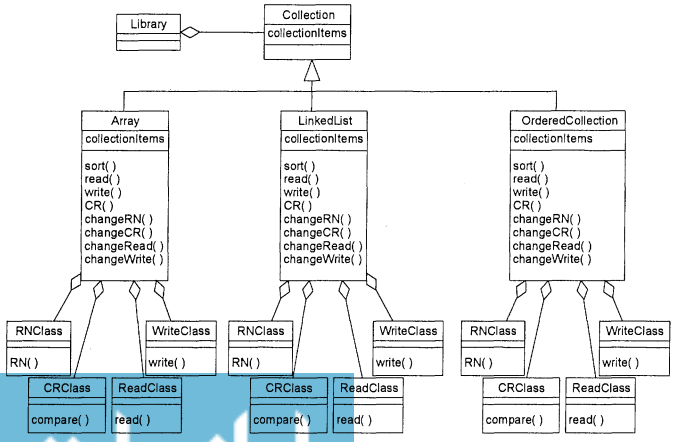
Adaptability Degree: 27
Performance Degree: 12



MODEL 6
RN, CR, read and write run-time adaptable. Sort is fixed.
Tuple:

$C_{ObjectAdaptLibrary}$ = *((CL, AD, Library), (CL, AD, Collection), (CL, AD, LinkedList), (CL, AD, OrderedCollection), (CL, AD, Array), (AT_m, AD, collectionItems), (OP_v, FX, sort), (OPv, AD, RN), (CL, AD, RNClass), (OP_v,, AD, Read), (OP_v,, AD, ReadClass), (OP_v,, AD, Write), (OP_v,, AD, WriteClass), (OP_v ,AD, CR), (CL, AD, CRClass))*

Adaptability Degree: 30
Performance Degree: 16 (measured value 1310ms)



PART 3

COMPONENTS

Chapter 9

APPLICATIONS = COMPONENTS + SCRIPTS

A tour of Piccola

Franz Achermann and Oscar Nierstrasz

Software Composition Group, IAM, Institute of Computer Science and Applied Mathematics, University of Bern, Neubrückstrasse 10, CH-3012, Bern, Switzerland. Email: {acherman, Oscar.Nierstrasz}@iam.unibe.ch, www: http://www.iam.unibe.ch/~scg/Research/Piccola/

Keywords: Composition Language, Composition Style, Forms, Context, Scripting, Piccola

Abstract: Piccola is a language for composing applications from software components. It has a small syntax and a minimal set of features needed for specifying different styles of software composition. The core features of Piccola are communicating *agents*, which perform computations, and *forms*, which are the communicated values. Forms are a special notion of extensible, immutable records. Forms and agents allow us to unify components, static and dynamic contexts and arguments for invoking services. Through a series of examples, we present a tour of Piccola, illustrating how forms and agents suffice to express a variety of compositional abstractions and styles.

1. INTRODUCTION

Piccola is intended to be a *general-purpose language for software composition*. Whereas existing programming languages appear to be suitable for implementing software components, and many scripting languages and fourth-generation languages have been developed to address the needs of particular component models, there has been relatively little work that attempts to develop a generalized approach that may span various architectural styles and component models.

We have argued elsewhere [1][24] that most object-oriented methods typically do not lead to pluggable component architectures (mainly because reuse is considered too late in the lifecycle) and that the resulting software

systems can be hard to maintain and understand because they do not make the run-time architecture explicit (the source code describes the classes, not the objects). To address this problem, we have proposed a conceptual framework for software composition that can be summed up as:

Applications = Components + Scripts

Components must conform to *architectural styles* [26] that determine the *plugs* each component may have (i.e., exported and imported *services*), the *connectors* that may be used to compose them, and the *rules* governing their composition. *Scripts* define specific connections of the components. Additionally, *glue* abstractions may be required to bridge architectural styles, and adapt components that have not been designed to work together, and *coordination* abstractions may be required to manage dependencies between concurrent and distributed components.

Piccola's runtime model consists of communicating agents. Scripts specify the behaviors of these agents. Agents invoke services and compose forms. Agents live in a *context* that contains the known services and forms for an agent. In this text we will show how components can be scripted in a declarative way by means of a *style* which defines a kind of "component algebra." Consider, for example, the well-known style of *pipes and filters:*

Table 1: Pipes and filters

Components:	File, Stream, Filter	*Files and Filters are external components*
Connectors:	<, |, >	*Three kinds of pipe operators*
Rules:	Filter < File → Stream	*A File piped into a Filter yields a Stream*
	Stream | Filter → Stream	*A Stream piped into a Filter is still a Stream*
	Stream > File → *nil*	*A Stream can be piped into a File*

Pipes and filters are "algebraic" in the sense that the composition of two components yields another component.

Unlike scripting languages that offer only a fixed set of compositional styles, Piccola allows you to *define your own styles* for different application domains. Rather than develop Piccola as an extension to an existing language, we felt it was important and necessary to emphasize a *separation of concerns* between component implementation and component composition. Our goal is to identify a well-founded set of features necessary and sufficient for specifying software compositions as scripts, while supporting an open-ended set of architectural styles. Piccola adopts a layered approach to achieve this goal. *External components* export services

transparently to each layer. For example, the abstract machine layer sees these services as ordinary channels and agents.

Applications	Components + scripts	*External components*
Architectural styles	Streams, events, GUI composition, ...	
Core libraries	basic coordination abstractions, basic object model	
Piccola language	Services, operator syntax, nested forms, built-in types	
πL abstract machine	agents, channels, forms	

The bottom level of the Piccola system provides an abstract machine in which *agents* asynchronously communicate *forms* through shared *channels*. This abstract machine implements the πL-calculus [13], a variant of the polyadic π-calculus [15] in which forms are communicated instead of tuples. The innovation at this level is the introduction of *forms*, which are immutable, extensible records (sets of bindings from labels to channels). Technically speaking, communicating forms rather than tuples does not alter the expressive power of the π-calculus, but it makes it much simpler to express higher-level abstractions in Piccola [25]. This simple foundation allows us to reason about complex and concurrent interactions using a well-developed formal model, and guarantees that the semantics of higher-level abstractions can always be precisely explained in terms of simple interactions.

The next layer defines the Piccola language syntax and semantics. We introduce *primitive values*, like numbers and strings, *higher-order abstractions* over agents, forms and channels, and *nested forms*. Abstractions and nested forms are defined simply by translation to the lower level model using hidden intermediate channels and agents. At this level we already begin to appreciate the expressive power of forms. Forms represent:

- Interfaces to components. Forms encapsulate a set of named services exported to clients.
- Arguments. Forms provide keyword-based arguments for services.
- Contexts. The static context represents all known services and components for any statement. The dynamic context collects services and capabilities that are passed from callers to callee.
- User-defined service.

As forms are immutable, operations on forms yield new forms with an enriched or reduced set of services. It is not possible to modify forms, thereby breaking by accident other agents using this form or component, but only to create new forms. We can see a form as a kind of "primitive object" with public and private features, but without any explicit notion of classes or

inheritance. More elaborate object models can be encoded directly in Piccola. Piccola permits form labels to be accessed as overloaded infix operators, which is convenient for expressing compositional styles.

The third layer defines libraries of basic composition abstractions, including control abstractions (e.g., if-then-else, try-catch), coordination abstractions (e.g., blackboards, futures), and other utilities, such as an interface to the Java world. The interface wraps Java objects and represents them as forms so that Piccola agents can access them.

At the fourth layer, libraries of architectural styles may be defined, such as push-flow or pull-flow streams, GUI composition, and GUI event composition. This is done by implementing connectors for such a style as infix operators on components. A style may also define coordination abstractions to manage interactions between components, and glue abstractions to adapt external components to a particular style, or possibly to bridge gaps between different styles [6][27].

Finally, application programmers can script applications using the connectors of a particular style and the glue abstractions to use external components.

This paper is structured as follows. The next section presents an example that illustrates the top-level view of a Piccola script. Then, in sections 3, 4 and 5, we present the Piccola language layer, and describe respectively, forms, agents and contexts. In section 6 we show how Piccola can be used to define a simple architectural style, and in section 7 we show how classes and mixins can be scripted. Finally, section 8 discusses related work and section 9 concludes this paper.

2. SCRIPTING COMPONENTS

In this section we present a small example of a Piccola script that uses styles for GUI composition and GUI event composition. The specification of event style itself is presented later in section 6. The reader should not worry too much about details of the mechanics of the script on a first reading, but pay attention instead to how Piccola is used to develop a high-level, declarative view of how this applications composed. The same application written directly in an object-oriented language would typically be more procedural, and emphasize low-level wiring of observers and observables [5]. The Piccola script, on the other hand, expresses the wiring by using compositional operators defined as library abstractions supporting an architectural style

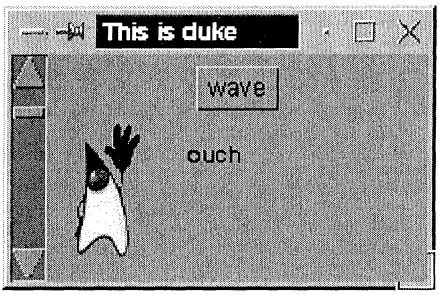


Figure 2: Duke scripted in Piccola

The script "`duke.picl`" in figure 2 uses an event style to wire the events and illustrates how the graphical layout is scripted. It also coordinates several agents. Running the script, a frame with Java's Duke appears (see figure 1). When we click on the `wave` button, duke waves at the speed controlled by the scrollbar on the left. When we click on Duke himself, he complains, issuing the message "ouch." After a short delay, the message disappears.

We now look at the individual parts of the script and identify the forms and agents when necessary:

1. We load a file "`nawt`" which defines several services we will use. The keyword **`root`** denotes a special form that represents the static context in which `duke.picl` is evaluated (see section 5). `load()` reads a set of definitions in a Piccola library script and returns a form containing those bindings. We then *extend* the static context by simply redefining **`root`** to be **`root`** extended by the result of `load()`.
2. Now our extended root context contains the service `awtComponent` defined in the loaded script. This service instantiates new AWT components and wraps them according to our style. We use it to create the duke component, a button, and a scrollbar. The form returned by `awtComponent` can be thought of as a kind of "primitive object" providing the service `set` (amongst others). This service allows us to send a form containing some properties. For example, we set the label of the `waveButton` component by invoking `set` with the argument form `Label = "wave"`. Note that `set` may be invoked either with a parameterized expression on the same line, or by passing an indented form on the subsequent lines. Either syntax can be used at any time. The arguments passed to `set` will cause these properties to be updated in the wrapped Java object. We do not change any default property of `duke`.

```
# File: duke.picl
# 1. load nawt services
root = (root, load("nawt")) # use event and AWT wrappers style

# 2. create AWT Components
duke = awtComponent("demos.duke.Duke")
waveButton = awtComponent("java.awt.Button").set(Label="wave")
speedScrollbar = awtComponent("java.awt.Scrollbar").set
    Minimum = 1
    Maximum = 800
    Value = duke.getSpeed()

# 3. do the event wiring
speedScrollbar ? Adjustment
    do: (duke.set(Speed = speedScrollbar.getValue()))
waveButton ? Action(do: duke.wave(val = 1))

# 4. click on Duke
counter = load("counter").newCounter(0)
sleep() = javaClass("java.lang.Thread").sleep(val = 2000)
duke ? MouseClicked
    do:
        duke.set(Message = "ouch")
        counter.inc()
        sleep()                    # sleep 2 seconds
        if (counter.dec() <= 0)    # if this was the last click
          then: duke.clearMessage()

# 5. arrange components in a panel
panel = newBorderPanel
    center = newBorderPanel
       north = Components + waveButton
       center = duke
    west = speedScrollbar

# 6. add panel into a frame and display it
exit() = javaClass("java.lang.System").exit(val = 0)
frame = awtComponent("java.awt.Frame").set
    Title = "This is duke"
frame.add(val = panel.java, type = "java.awt.Component")
frame ? WindowClosing(do: exit())
frame.pack()
frame.show()
```

Figure 2: Duke script

3. Next, the events are wired using a compositional notation with infix operators, (see table 2). The style defines a set of *event types*, like `Adjustment` and `Action`. Each event type is modeled as an abstraction that takes a *response* (a form containing a `do` service) as a parameter and yields a *listener*. The resulting listener may be bound to a component with the infix `?` operator.

Table 2: GUI event composition style

Components:	C	*GUI-Component*
	E	*Event type*
	R	*Response*
	L	*Listener*
Connectors:	(), ?	
Rules:	E(R) → L	*Compose an event type with a response to get a listener*
	C ? L → ()	*Connect a component to a listener*

For example, when the scrollbar is adjusted, the new speed value is set in the `duke` component, whereas clicking on the button causes duke to wave.

4. When we click on duke, he displays a complaining message. The message disappears after a short delay. Each time the user presses the mouse on duke (`MouseClicked`) an agent runs the code given in the response. We do not see the agent directly, but we specify the `script (do: duke.set(Message="ouch", ...)` he executes. The agent runs in a context that contains bindings for the forms `duke` and `counter`, as well as the services `sleep` and `if`.
   Note that the bindings returned by `load("counter")` are not used to extend **`root`**. We directly use the exported service `newCounter()` to construct a thread-safe counter.
5. The graphical layout uses a different composition style from the event wiring. We use the service `newBorderPanel` exported by "`nawt.`" We define a new panel by invoking service `newBorderPanel`, which creates a new Java panel with a border layout manager. The argument is a form specifying sub-components with constraints north, south, west, east, or center, according to the border layout manager of Java [7]. A sub-component may itself be an instance of `newBorderPanel` or even a list of components. In this case these components are arranged using a flow layout in an inner panel.

Table 3: GUI Composition style

Components:	C	*GUI-Component*
	List	*List of Components*
Connectors:	+, newBorderPanel	
Rules:	List + C → List	*Builds a new list with additional element*
	List + List → List	*concatenate lists*
	NewBorderPanel(Form)	*layout Components in the form*

This determines the stretching properties of the sub-components. Component lists are built up by starting with an empty list (i.e. `Components`) and adding widgets using the + operator. Glue code maps the interfaces of Java objects to fit the style. Note that GUI composition in Piccola using an appropriate style is more declarative than what one would typically write in a conventional object-oriented language. Contrast it with the code fragment necessary to achieve the same layout in Java:

```
Panel panel = new Panel(new BorderLayout());
Panel InnerPanel = new Panel(new BorderLayout());
Panel buttons = new Panel();    // using the default
flow layout
buttons.add(waveButton);
innerPanel.add(buttons, BorderLayout.NORTH);
innerPanel.add(duke, BorderLayout.CENTER);
panel.add(innerPanel, BorderLayout.CENTER);
panel.add(speedScrollbar, BorderLayout.WEST);
```

6. Finally, the panel is put into a new frame, which is displayed. As the Piccola AWT style uniformly wraps AWT components from Java, we can use methods `pack()`, `show()` etc. directly from the underlying peer Java objects.

This simple example illustrates several important points about Piccola:

- Piccola syntax is extremely lightweight. There are only four keywords (**`root`**, **`dynamic`**, **`def`** and **`return`**) and six reserved operators.
- Forms are ubiquitous in Piccola. They are used to represent interfaces to components, arguments for services, and contexts for agents.
- Although Piccola is not designed as a Bean scripting language, one can use it to compose Beans — or any other kinds of components, for that matter — by defining a suitable architectural style.
- When styles are defined as "component algebras," the resulting scripts are highly declarative and make the wiring of components explicit.

- In the next three sections, we give an overview of all the features of Piccola, namely that of forms, communicating agents, and contexts.

3. WHAT IS A FORM?

We have identified forms as a central concept needed for composition. A form is a mapping of labels to values. The empty form has no labels. Forms in Piccola are themselves values and may therefore be nested. Many data-structures have a natural embedding as forms. Forms are written as sequences of *bindings*, separated by commas or new-lines and structured using brackets or indentation:

```
baseForm =
   Text = "foo"
   Name = Text
   Size = (x = 10, y = 20)
```

The form `baseForm` contains three labels: `Text`, `Name`, and `Size`. The nested form `baseForm.Size` has labels `x` and `y`. *Projection* is used to fetch elements of a form. For example, the projection `Form.Size.x` yields 10.

Forms are built as a sequence of bindings. Each individual binding is added to the form it follows. At the same time, each binding also acts as a declaration for subsequent code. Thus, the identifier `Text` in the binding `Name` is bound to the string `"foo"` in the previous line. Forms and sequences of statements are unified in Piccola. The whole assignment defines a nested form bound to the label `baseForm` in the global form **`root`**.

3.1 Extending Forms

New forms can be built by *extension*. A form, or more precisely the list of its bindings, may be concatenated with other bindings, which yields a new form. We can extend `baseForm` with a binding for `Color`:

```
coloredForm =
   baseForm
   Color = "green"
```

Now the `coloredForm` has a label `Color` in addition to the labels of `baseForm`. We cannot detect in the extended form how and in what order the labels where added. Note that `baseForm` remains unchanged.

Bindings may also be overridden by new bindings. Clients using an extended form will only have access to the most recent binding for a label. The following example makes a new form with a modified `Size`:

```
modForm =
   baseForm
   Size = (baseForm.Size, x = 15)
```

This extension makes only minimal assumptions on the labels in `baseForm`. It only assumes the presence of label `Size` in `baseForm`. We add a binding for a new `Size`. The new size itself is an extension of `Size` in the original form with a overridden label `x`. Note that this extension would also work if the original size would contain different labels, say for example three parameters `x`, `y`, and `z`. Then, our modified form would also contain these bindings with a modified `x` value. We heavily use this feature of forms in building reusable abstractions.

It is also possible to extend one form by another, rather than just specifying individual labels to bind. This is an easy and compact way to have default parameters:

```
withDefaults =
    Font = aSystemFont
    baseForm
```

Now, we can project on `Font` in the form `withDefaults`. If `baseForm` already contains a binding for the label `Font`, this value is returned, otherwise the value `aSystemFont` is returned.

Projecting on an unbound label is a type error and yields an undefined value. (Using this value generates an exception.) Type systems for πL and Piccola have been explored [13] but are not presented in this paper.

3.2 Services

In Piccola, we represent everything as a form. Literal values like strings or numbers are forms in the same way strings and numbers are objects in pure object-oriented systems like Smalltalk. Forms are used to encapsulate sets of services. Services themselves are also represented as forms. A service can be invoked with a function-call syntax, but is actually a form with a hidden label that gives access to an agent that represents it. (We use the term *service* rather than "function" to emphasize the fact that the invoked behavior is provided either directly or indirectly by an external component.)

As everything is represented as a form, the arguments for invoking services are also forms. Therefore, they have in general only one argument.

```
hello() =
     println("hello world")
```

This statement defines a service and assigns it to the form `hello`. The body of the service consists of a call to another service: `println`. When `hello` is invoked, it returns whatever `println` will return.

An alternative can be used when no formal parameter is needed. We can omit the brackets and write:

```
hello: println("hello world")
```

The colon signals that the right hand side is an abstraction. The colon notation sometimes makes code easier to read. Drawing from our earlier example in section 2, the following two forms are strictly equivalent in Piccola:

```
do: duke.wave(val = 1)
do() = duke.wave(val = 1)
```

To see that a service is just a form, consider the following, equivalent statement:

```
hello = \() = println("hello world")
```

Here, the label `hello` is bound to the anonymous abstraction `\() = ...` Anonymous abstractions are sometimes convenient for defining coordination abstractions, but we will rarely use them directly. Most of the time, a form with a `do` service is more convenient to use.

External components export primitives services to Piccola, but higher-level services can be scripted in Piccola. We therefore speak of the body of a service as its *script*. For example, the script of the `hello` service above is

```
println("hello world").
```

When a service is invoked, an agent evaluates its script (also a form). The `root` context this agent runs in provides access to statically bound services (like `load`) and a dynamic argument that gets passed at invocation time.

We can extend services like any other form and, for example, add labels documenting their interface. Piccola makes no assumption about such additional labels.

```
myhello =
   doc = "My hello world"
   hello
```

There are several ways to invoke services. The argument form can be enclosed in brackets or given by indentation. The following alternatives all invoke a *higher-order* service `if`. When it is invoked with a boolean value as an argument, it returns a service taking as argument a form containing labels `then` or `else`.

```
if (name == "main")
   then: hello()
if (name == "main") (then: hello()  # a one liner!
branch = if(name == "main")         # curried: apply
branch                              # boolean branch is a
                                    # service:
   then: hello()                    # apply cases
```

As services are first class values, we could also directly bind `hello` to the label `then`:

```
if (name == "main")
   then = hello                     # bind then to (form)
                                    # hello
```

Boolean values are encoded as forms that provide a `select` service. This service either selects a true or false binding of its argument:

```
true = (select(B) = B.true)
false = (select(B) = B.false)
```

Services in Piccola always take a single form as an argument. Since services are values, however, it is possible to define curried services (i.e., taking a single argument and returning a service). Consider the implementation of `if` as it is used above:

```
if(Boolean)(Cases) =                    # curried: same as:
                                        # if(B) = \(C) = ...
    withDefaults =
       then: ()
       else: ()

       Cases
    Case = Boolean.select               # select a case
       true = withDefaults.then
       false = withDefaults.else
    return Case()                       # evaluate branch
```

The service takes two forms as its arguments: `Boolean` and `Cases`. In the body of the service, we first provide `Cases` with default `then` and `else`. The defaults we supply are dummy services that return the empty form, written as `()`. Next, we use the boolean to select either the `then` case (the boolean is true) or the `else` branch. Finally we evaluate the case selected and return it as the result of the `if` service.

What would happen if we omitted the **`return`** keyword in the above definition? Then the result of an application `if(B)(C)` would be a form containing not only the bindings returned by `Case()`, but also those of `withDefaults` and `Case`! The use of the keyword **`return`** ensures that only the value of the expression that follows is returned. All prior bindings are strictly local. This same mechanism can be used to build objects with private and public features.

3.3 Operators

Piccola supports user defined operators. Any sequence of operator characters like `-`,`+`,`*`,`=`,`!`,... represents an infix or prefix operator. As is usual in object-oriented languages supporting infix operators, such operators are treated as projections on their left-hand side component with the right-hand side component as the argument. The label associated with the operator token has two underscores for infix and one for prefix-use in front of it. For instance: `name == "main"` is interpreted as `name.__==("main")`. Identifiers may also be infix operators when they are enclosed in single backquotes as in `5 `mod` 3` which is `5.mod(3)`. Similar: `- 4` is interpreted as `4._-()`. Sequences of infix terms associate to the left, i.e. `a | b | c` is `(a | b) | c` or, equivalently, `a.__|(b).__|(c)`.

Infix operators are used to syntactically present architectural styles in a more compositional or algebraical way, as illustrated by the example in section 2.

3.4 Scopes

So far we have only seen simple bindings of labels to expressions using labels bound in previous statements. The right-hand side of a binding can never refer recursively to the label being bound. In practical applications, however, we often need recursive services and forms. The keyword **def** defines such a binding. In definitions, the right-hand side can refer to the identifier being assigned to, provided it is used within an abstraction:

```
def fact(N) =
   if (N < 2)
      then: 1
      else: N * fact(N-1)
```

While **def** is not surprising for services, we also use it to construct fixpoints for plain forms. In this circumstance it allows us to define forms with a notion of self:

```
def cout =
   __<<(X) =
      print(X)
      return cout
nl = "\n"
cout << "Hello World" << nl
```

Evaluating the term `cout << X` prints `X` and returns `cout`. Therefore, we can write sequences of such terms as in C++.

Note that in each of these examples the recursion occurred within an abstraction. The following examples, by contrast, are not sound in Piccola:

```
def silly = (a = silly)
def sillier = sillier
```

and result in run-time errors. The agent that builds the fixpoint reads it before it is correctly set. The following service is uninteresting, but sound:

```
def sillyButOK() = sillyButOK
```

The **def** keyword can also be used to define mutually recursive services. When two or more services should refer each other, they can be enclosed in a common, recursive scope:

```
def myscope =
    a() =
        ...
        myscope.b()                    # call b in myscope
    b() =
        ...
        myscope.a()
```

Note that we could equally omit `myscope` in the body of service `b()` to call to `a()`.

4. COMMUNICATING AGENTS

The semantics of Piccola is given in terms of communicating agents. There are two predefined abstractions necessary to control these agents: one to asynchronously evaluate a `do` service by a new agent and one to synchronize running agents.

The `run` primitive evaluates the `do` service of a form as a separate agent. The result of `run(...)` is the empty form. This result is returned in parallel to starting the new agent. The term `newChannel()` creates a new channel. Channels provide atomic send and receive services to communicate forms. The sender cannot detect when and whether the value sent is received by a communication partner. Receiving a value from a channel blocks unless someone has sent a form to it. If one or more forms are sent, then an arbitrary one of them is received. There is no ordering on the values communicated along a channel. The following script creates a channel `ch` and starts two agents that communicate a form along it:

```
ch = newChannel()
run (do: ch.send("a form"))
run
    do:
        v = ch.receive()
        println("I received " + v)
```

Running this script, the second agent will print out `I received a form`. The library script "`pil`" provides a style that makes programming with channels and agents more convenient, and mimics the operators of the lower-level pL machine. The script redefines `newChannel` and equips new channels with infix operators `!`, `?` and `?*` instead of `send` and `receive`. The operator `?*` attaches a "replicated agent" to the channel.

Table 4: pil-style

Components:	C	Channels
	A	Agents
Connectors:	!, ?, ?*	Output, input, replicated input
Rules:	C ! Form → A	Send form along channel C
	C ? Abstraction → A	Receive form and run abstraction
	C ?* Abstraction → A	Multiple receive from channel.

A replicated agent behaves like an endless supply of agents, always ready to receive another message. These operators send and receive forms in their own agents. Using the pil-style, the above script becomes:

```
root = (root, load("pil"))      # redefines newChannel
ch = newChannel()
ch ! "a form"                   # send the string
ch ? \(v) =                     # receive a value, then
   println("I received " + v)   # run the service
```

The two predefined abstractions `run` and `newChannel` are enough to recover the expressive power of pL. For example, a stop service can be implemented as:

```
stop() =
    newChannel().receive()      # will never receive
                                # anything
```

Calling `stop()` will never return and therefore stop the client agent.

Another useful concurrency abstraction is one that evaluates two abstractions in parallel. It returns the result of one of the two passed abstractions. When both abstractions terminate, either result is returned. However, when we know that only one branch terminates and the other stops, the result of `OrJoin` is uniquely determined:

```
OrJoin(X) =
   ch = newChannel()
   run (do: ch.send(X.left()))
   run (do: ch.send(X.right()))
   return ch.receive()          # blocks unless there is
                                # one result
```

Here, we run two agents in parallel. The two agents execute the left and the right abstraction given. The service `ch.receive()` blocks, unless one value is sent on it. Once a value is sent to the channel, this value is returned.

In the next section, we will use these services to implement an exception handling mechanism within Piccola.

`OrJoin` and `stop` are examples of coordination abstractions. For example, `OrJoin` is used to coordinate two agents such that only one agent returns a result.

5. CONTEXTS

When an agent evaluates a script, it may make use of services defined in the current context (or "environment"). Piccola models contexts explicitly as forms. Since contexts are therefore first-class values, one can implement various abstractions to support modules and packages. In contrast to Piccola, most languages provide a predefined and fixed way to import modules and look up imported services.

The special form **`root`** denotes the (static) context in which identifiers are looked up. Instead of writing:

```
print("Hello")
```

we could equally say:

```
root.print("Hello")
```

Similarly, bindings also extend the **`root`** form for subsequent statements. It is also possible to assign any form as new root or to use root as an ordinary form. For example, `load()` locates a script and evaluates it. It returns the form defined by the script. Assume we have a script `"hello.picl"` with the contents:

```
# File: hello.picl
hello: println("This is the hello script")
```

We can now import the bindings into the **`root`** and use `hello` directly:

```
root = (root, load("hello"))          # extend our root
                                      # with hello
hello()                               # call hello
```

or we can load the script and keep it in a separate form. This prevents cluttering up our **`root`** namespace:

```
x = load("hello")                    # bind hello to x
x.hello()                            # and use it
```

When the Piccola run-time system is initialized, `root` contains the services of the basic Piccola composition abstractions.

5.1 Dynamic Contexts

Statically compiled languages typically use static (lexical) scoping whereas dynamically compiled and interpreted languages often use dynamic scoping or a combination of static and dynamic scoping. Piccola is statically scoped, but offers *dynamic scoping on demand*. Although static scoping is good enough for most purposes, it turns out that certain kinds of coordination and control abstractions are next to impossible to define without dynamic scoping.

As an example, consider exception handling. Most languages that provide exception handling as a built-in construct allow an exception to be raised in the context of some service provider, and thereby cause an associated exception handler of the client to be invoked. In languages that do not provide exception handling, it can be very difficult to simulate. Let us see now how such an abstraction can be defined in Piccola by explicitly passing dynamic contexts between agents.

An example application is the `import` service, which is defined as:

```
import(F) =
   x = findFile(F.name)
   if (isEmpty(x))
      then: raise("Cannot locate Script: " + F.name)
   # otherwise x points to a valid file. We return its
   # contents:
   return try
      do: builtinLoad(x)(F.context, scriptLocation = x)
      catch(E):
         raise("Error in Picclet " + x + "\n" + E)
```

Import tries to find a given file. When this file cannot be located, it raises an error. Otherwise, the location `x` is read and executed. The service `builtinLoad` loads, parses, and executes the script at location `x`. It is possible that this process raises an error. This error is caught and reported to the user. The service `builtinLoad(x)` returns a anonymous abstraction containing the script at `x` as its body and `root` as its argument. We invoke this context with the context passed (`F.context`) extended with the location

of the script itself. When `builtinLoad` returns successfully, `import` returns the contents of the file.

Observe that `try` and `raise` are normal abstractions, whereas `do` and `catch` are ordinary labels in the argument to `try`. Here are the implementations of `try` and `raise`:

```
try (block) =
   exception = newChannel()
   result = OrJoin
      left:
          e = exception.receive()
          return block.catch(e)
      right:
          raise(e) =              # define a local raise
                                  # abstraction
             exception.send(e)
             stop()
          dynamic = (dynamic, raise = raise)
          return block.do()
   return result

raise(E) =                        # use dynamic raise
   dynamic.raise(E)
```

Let us first look at the body of `try`. It creates two agents and waits for one of them to terminate. We have already seen `OrJoin` and `stop` in section 4. The `right` agent runs the `do` service of the argument to `try`. This service may terminate normally, causing the agent to return a result, or it may raise an exception, and transfer control to the `left` agent. The `left` agent blocks and waits if an exception is raised. If so, it evaluates the `catch` service of the argument to `try`. Otherwise it does nothing.

The difficulty here is that the client's `do` service knows nothing about the exception channel we want to use to coordinate the two agents. The solution is to define a *local* `raise` abstraction that will signal the exception and stop the `right` agent. This `raise` abstraction is injected into the dynamic context made available to the `do` service. When the `do` service calls the *global* `raise` abstraction, it in turn calls the dynamic one, and the right thing happens.

Whenever a service is called in Piccola, the form **`dynamic`** is passed implicitly together with the actual parameter. If the client has extended its dynamic context with any additional services, these will then be available to the called abstraction.

5.2 Passing the Dynamic Context

For readers with some background in the π calculus, it may be helpful to have a closer look at how services are invoked. For that purpose, we show

the protocol that is used by service invocations. This protocol can be implemented nicely on top of Piccola using agents and channels. A service becomes a channel together with a replicated agent that implements its body and returns a result. An invocation consists in communicating a dynamic context to this agent along the service-channel. This context will contain the argument (`args`) and a result channel. The replicated agent will send its result along that result channel.

```
root = (root, load("pil"))          # redefines

                                    # newChannel
fact = newChannel()                 # the service channel

fact ?* \(Dynamic) =                # the service body...
   N = Dynamic.args                 # Assign argument

                                    # form

   if (N > 1)                       # factorial:
      then:
          # invoke fact(N-1):
          h = newChannel()          # the result channel
          fact ! (Dynamic, args = (N - 1),result = h)
          h ? \(Result) =

              Dynamic.result ! (N * Result)

      else:

          Dynamic.result ! 1
```

Note that we use our previously mentioned pil-style. In the code, we use the identifier `Dynamic` instead of the Piccola keyword **`dynamic`**. Observe the invocation of `fact(N-1)`:

- We first create a reply channel `h`.
- We then send an invocation to the service channel (`fact`). The invocation consists of the context for the agent responsible to evaluate the service. The context at least contains the argument form and the result channel.
- We receive the result on the reply channel h. Once the service agent delivers a result, we fetch it and continue.

An invocation closely corresponds to the responsibilities the agent implementing the service has. The service is modeled by a replicated agent receiving invocations. An invocation consists of a form. The arguments are by convention bound by label `args`, the result channel is bound by label `result`. The result is returned by sending it along the result channel, from where the client will pick it up.

6. IMPLEMENTING STYLES

This section presents the implementation of the event composition style used in section 2. Participants transmit or receive pieces of information in response to events. Components that emit events are called informers, those that receive them are called listeners [2].

We show code to glue the services provided by objects of the Java AWT Event framework to the event composition style of table 2 that can be used as:

```
javaComponent ? EventType(Response)
```

The `?` with a given event type connects a `Response` to an event within the Java component. A `Response` is a form with a `do` service.

6.1 Interfacing to Java Components

The low-level bridge to Java objects from Piccola is done using the predefined abstractions `javaClass` and `javaObject`. These generic glue abstractions create Java objects and return forms giving access to the public methods of them. The methods are invoked like any other service but the arguments are given as nested forms with labels `val` or `val0`, `val1`, `val2`, etc. since arguments for Java are tuples instead of being keyword based. For overloaded methods, we must also give the type of the arguments in order to select a unique method implementation in Java.

The Piccola Java interface also provides some generic listener classes, like the class `pi.piccola.bridge.GenericActionListener`. These classes allow us to call Piccola services from Java. The generic action listener class, for example, implements the Java interface `java.awt.event.ActionListener`. An action listener that prints the events is created by:

```
LC =
javaClass("pi.piccola.bridge.GenericActionListener")

listener = LC.new
    val = dynamic
    val1 = (actionPerformed = println)
```

The constructor for the listener class requires two parameters, the first is the dynamic context which will be passed to the listener service, in case the listener service makes use of services in the dynamic context. We need to pass this context explicitly, since Java does not offer a notion of context. The second parameter contains an abstraction to which the event is delegated. The handler for action listeners must be bound by the label

`actionPerformed`. The Java constructor for `GenericActionListener` is given as:

```
public GenericActionListener(Form context,Form delegate);
```

A listener object may be plugged into components using `void addActionListener(java.awt. event.ActionListener)`. An event is then forwarded to the service `actionPerformed` within the dynamic context passed. For example, the listener can be added to a button:

```
button = javaObject("java.awt.Button")
button.addActionListener(val = listener)
```

6.2 The GUI Event Composition Style

To support the GUI event composition style, we need to (1) model event types as abstractions that take `do` services as arguments and return listeners, and (2) extend GUI components with a `?` operator to attach listeners. For example, the following code creates a listener for `Action` events and attaches it to a Java Button that has been wrapped to conform to the style.

```
myButton = awtComponent("java.awt.Button")
myButton ? Action(do = println)
```

Since there are many different types of event in the AWT framework, we use a generic glue abstraction, `newEventType`, to instantiate event types for our style:

```
Action = newEventType
   genericListenerClass =
javaClass("pi.piccola.bridge.GenericActionListener")
   listenerMethod(service) = (actionPerformed=service.do)
   addListener(Component) = Component.addActionListener
```

The argument to `newEventType` is a form with the following labels:

- `genericListenerClass` is a factory service to instantiate Java listener objects. These objects will be created using `new()` with arguments `val0` for the dynamic context and `val1` for the delegate form.
- `listenerMethod` is a service that returns the delegate form used to instantiate the generic listener class.
- `addListener(Component)` is a (curried) service encapsulating the

method to add listener instances.

Here is the implementation of `newEventType`. Note that it is a curried service — the event type it returns (e.g., `Action`) is itself a service that will return a listener. A listener provides a `register` functionality that will be used by GUI components:

```
newEventType(P)(Response) =
   register(Component) =
      ConstrArgs =
         val = dynamic
         val1 = P.listenerMethod
             do(E):
                Response.do
                   Informer = Component
                   Event = E
      listener = P.genericListenerClass.new(ConstrArgs)
      P.addListener(Component)(val = listener)
```

The `listener` object is instantiated using the `new` service of the (passed) generic listener class. As expected, the argument form for `new()` is the current dynamic context and a form with the delegate services, e.g. a binding `actionPerformed` for the action event type. Finally the listener registers itself on a Component by delegating registration requests to `addListener()`.

The glue abstraction `awtComponent` instantiates AWT objects and extends them with the `?` operator. This operation uses double dispatch to register the listener `L`:

```
awtComponent(ClassName) =
   object = javaObject(ClassName)
   def self =
      object
      java = object
      set(P) = ...                    # set properties P
      __?(L) = L.register(self)       # pass the component

    return self
```

The Java class is instantiated, and the Piccola representing it is extended with services needed to support the event style. In addition, the original base object is still available by a projection on the label `java`.

The implementation of this style may seem somewhat convoluted, but this is largely a side-effect of the fact we are adapting an object-oriented interface to a more compositional style. Keep in mind that the code presented here needs to be written only once. It can then be exploited by any number of scripts. Furthermore, advanced features like `dynamic` contexts are

typically used only to implement abstractions to support a particular style, and do not normally appear in top-level scripts.

7. SCRIPTING CLASSES

Although Piccola has no predefined object model, it is possible to implement different object models on top of it, much in the same way that CLOS is defined on top of Common Lisp [8]. In this section, we use one such model to script classes and mixins [3]. This particular model is implemented by a `Class` abstraction and a initial class `Object`, from which all classes inherit. The following code loads the object model and creates a `Point` class:

```
root = (root, load("classes"))   # get Class, Object
Point = Class
   name = "Point"
   super = Object
   instanceVars: (x=newRefcell(), y=newRefcell())

   delta(P):
      asString() = "x = " + P.self.x.get() +
                   ", y = " + P.self.y.get()

      rep() =
         println(P.self.class.name + ".new(" +
            P.self.asString() + ")")
    initialize(Init) =
          P.self.x.set(Init.x)
          P.self.y.set(Init.y)
```

We use the abstraction `Class` to create a new class. Individual classes are parameterized by the following bindings:

- The `name` of the class.
- The `super` or parent class from which this class is derived. The model described here only supports single inheritance.
- A service `instanceVars()` that creates the additional instance variables for instances of this class. Each instance variable is represented by a reference cell with `set` and `get` accessor services. The service `instanceVars` is optional. The default binding for this parameter assumes that there are no new instance variables to be added.
- The `delta(P)` abstraction defines the differences of the new class with respect to its super class. The formal parameter `P` contains the nested forms self and `super` for self sends and super calls. The `Point` class defines three methods: `rep()`, `asString()` and `initialize()`. The initialize method is special: whenever we override this method, a call to

the overridden `initialize()` is inserted before the overriding method. We can omit a call to `super.initialize()`. This behavior is implemented in the `Class` abstraction.

The abstraction `Class` creates forms with a service `new()` to create and initialize new objects. For instance, a point is created by:

```
aPoint = Point.new
   x = 1
   y = 2
```

Calling `aPoint.rep()` prints out the string: `Point.new(x = 1, y = 2)`, as expected.

Whenever a new instance is created, `delta()` and `instanceVars()` of all subclasses in the inheritance chain starting from `Object` are called. The assembling is done within a scope definition for `self`. That way we pass `self` and the intermediate objects as `super` to each call to `delta()`. Once the object is built `initialize()` gets called to establish the invariant of the object.

Having the instance variables created by `instanceVars` is not a restriction of the object model. In fact, we could also create the instance variables directly in `delta()`:

```
Point = Class
   name = "Point"
   super = Object
   delta(P):

      x = newRefcell()
      y = newRefcell()    ...
```

But keeping them by in separate intention-revealing parameter for classes makes the code more self-documenting. In addition, clients that stick to `instanceVars()` for creating instance variables can implement generic operations for cloning objects or inspecting facilities.

`ColoredPoint` is a subclass of `Point` with an additional `color` field and overridden method `asString()`:

```
ColoredPoint = Class
   name = "ColoredPoint"
   super = Point

   instanceVars: color=newRefcell()
   delta(P):
      asString() =
         P.super.asString() + ", color = " +
P.self.color.get()
      initialize(Init) =
         P.self.color.set((color ="Black",Init).color)
```

The method `asString()` overrides `asString` of the `point` class and appends a representation for the color of a point. Note how form extension is used to initialize the `color` slot with a default value.

Mixins are classes with a free `super`. Mixin-composition composes two mixins to a new one. Applying a mixin to a class yields a new class. A `color` mixin may look as:

```
ColorMixin = Mixin
   name = "Colored"
   instanceVars: color=newRefcell()
   delta(P) = ...                          # as above
```

This mixin adds a color part to any class it is applied to. Note that the parent class is not specified here. Now, we can apply the mixin to our previous class:

```
myClass = ColorMixin * PointClass
point = myClass.new
   x = 1
   y = 1
   color = "Yellow"
```

Note that we use the flexibility gained from the keyword-based argument to initialize the reference cells. We just pass a form as initializer, each `initialize()` method needs only its specific arguments. The `Mixin` abstraction builds a class name by prefixing the name of the mixin (e.g. `"Colored"`) to the name of the parent class (e.g. `"Point"`). Another mixin may add a `move()` method to change `x` and `y` coordinates of a given point:

```
MoveMixin = Mixin
   name = "Moveable"
   delta(P) =
      move(Diff) =
         P.self.x.set(Diff.x + P.self.x.get())
         P.self.y.set(Diff.y + P.self.y.get())
moveablePoint = ColorMixin * MoveMixin * PointClass
```

Observe that `ColorMixin * MoveMixin` is also a mixin. We summarize the classes and mixin style.

Table 5: Classes and mixins

Components:	Class, Mixin	
Connectors:	*	mixin operator
Rules:	Mixin * Class → Class	Mixin application
	Mixin * Mixin → Mixin	Mixin composition

The `Class` and `Mixin` abstractions shown in this section are implemented by approximately 80 lines of Piccola code. This illustrates that it is possible to encode a useful inheritance composition mechanism with feasible effort. Schneider [25] has shown how to encode other forms of inheritance composition, like Beta-style [10] composition.

8. RELATED WORK

In the past years, there has been considerable work on the foundations of concurrency, and much of this on process algebras and process calculi. The π-calculus [15] has proven to be successful for modeling concurrent objects [22][23][29]. The πL-calculus [13] replaces tupel communication of the polyadic π-calculus with monadic form communication.

Pict [20] is a language that builds on the polyadic asynchronous π-calculus. Pict's language constructs are provided as syntactic sugar on top of the core calculus. We have used Pict to run extensive experiments with different object models [11][23] and synchronization policies [28] as examples for composition mechanisms. These experiments led us to conclude that form-based communication would be a better basis for modeling composition than tuple-based communication, and led us to develop the πL-calculus [12][13]. Pict was developed to study the relation of types and concurrent programming, whereas Piccola is used to experiment with composition abstractions.

We have been experimenting with different variants of Piccola. The version described here is Piccola 2.0. It completely hides the πL-primitives of the underlying process calculus as services, whereas these operators are visible in other versions. Piccola 2.0 can be compared to functional languages, where concurrency primitives where added, like this is done in CML [21]. In another variant, Piccola(T), we experiment with a type system for the πL-calculus [13]. Piccola(T) reflects the πL-operators as language primitives as in Pict. The type system is sound and complete, but lacks parametric polymorphism, which would be needed to type generic abstractions. We have also worked on extending the πL-calculus to the Form-calculus, which supports additional operators to hide labels. Piccola(F) offers these restriction operators as primitives [25].

In a much earlier paper with a similar title, we have explored visual composition of objects using scripts [18]. The present work provides a concrete textual syntax and a formal semantics for scripts.

The syntax of Piccola deliberately resembles that of Python [14][30]. Python is an object-oriented scripting language that provides a simple integration of functions and objects. Python models objects and classes in

terms of dictionaries (which resemble forms). Methods and functions can be called either with positional parameters (i.e., tuples) or with keyword arguments (i.e., à la forms). Python provides operator overloading, and can also be used to implement architectural styles much in the way described in this report. It provides limited support for reflection, and it is possible to change the underlying object model to a certain degree (though Python does not have a meta-reflective architecture like Smalltalk). Python is not inherently concurrent, though there is a Posix-dependent threads library, and some researchers have experimented with active object models for Python [19].

In Perl [30], procedures may specify the visibility of their local variables in its declaration. To the best of our knowledge, Piccola is the first language that offers both static scoping and the possibility of dynamic scoping on demand, within a formal framework.

Aspect-oriented Programming [9] is an approach to separating certain aspects of programs that cannot be easily specified as software abstractions. AspectJ is a language used to specify aspects that can be weaved into Java source code. Initial experiments have shown that certain aspects can be nicely expressed in Piccola. For example, Readers and Writers synchronization policies cannot be factored out as software abstractions in Java whereas this is relatively straightforward in Piccola. Whether aspects in general can be addressed by Piccola's compositional paradigm of agents and forms is an open question.

Coplien uses C++ as multi-paradigm language [4]. He uses C++ built-in paradigms like OO-inheritance or templates to match different component models and styles as they evolve from domain analysis.

9. FUTURE WORK AND CONCLUSIONS

We have described how Piccola supports the paradigm that Applications = Components + Scripts. We show how components conforming to a style are scripted and how different styles can be implemented within Piccola. This leads to a layered approach, where the abstractions provided by one layer connect components of the next level in a more declarative way.

We use forms to represent components, scripts, services, arguments to services, glue and coordination abstractions, and static and dynamic contexts. For an open component approach, however, it is clear that we must be able to cope with components obtained at run-time, possibly through network middleware. In this case Piccola must provide some reflective capabilities. It is not yet clear what capabilities precisely are needed to inspect forms. Should labels be first class values or is it enough to check for

the existence of a given binding in a form? We are currently investigating lightweight approaches, like providing built-in abstractions to iterate over all labels of a form. This allows us to define more generic wrappers for forms, but forbids introducing *new* labels.

Another issue related to open systems is distribution. It is not yet clear whether the notion of locality should go into the channels, (as for example in Klaim [17]) or whether it should be handled by providing dynamic services.

A flexible type system is needed to cope both with statically known components as well as dynamically introduced ones. Should the type system be defined at the level of the πL-calculus (as is the case in Piccola(T)) or at the Piccola language level? Can we develop a type system that captures whether a service returns, may raise an exception, or block? Instead of a type system, could we augment Piccola with an *assertion language* that would allow us to express and reason about the *contracts* that components require and ensure, and correspondingly about the properties guaranteed by an architectural style? Other important non-functional properties include safety and security, real-time properties and reachability. For example, what services are needed by a composition environment such that we can safely install, upgrade, and de-install components without breaking other parts of the system?

Piccola is available from
`www.iam.unibe.ch/~scg/Research/Piccola/`

ACKNOWLEDGEMENTS

The authors thank the members of the SCG, especially Jean-Guy Schneider, Serge Demeyer, Sander Tichelaar, and Markus Lumpe for helpful comments on improving this paper.

This work has been funded by the Swiss National Science Foundation under Project No. 20-53711.98, "A framework approach to composing heterogeneous applications".

10. REFERENCES

1. Franz Achermann, Markus Lumpe, Jean-Guy Schneider and Oscar Nierstrasz, "Piccola - a Small Composition Language," *Formal Methods for Distributed Processing, an Object Oriented Approach*, Howard Bowman and John Derrick. (Ed.), Cambridge University Press., 2000, to appear.
2. Daniel J. Barrett , Lori A. Clarke, Peri L. Tarr and Alexander Wise, "A Framework for Event-Based Software Integration " *IEEE Transactions on Software Engineering*, vol. 5(4) , October 1996 , pp. 378-421 .

3. Gilad Bracha and William Cook, "Mixin-based Inheritance," *Proceedings OOPSLA/ ECOOP'90, ACM SIGPLAN Notices*, vol. 25, no. 10, Oct. 1990, pp. 303-311.
4. James O. Coplien, Multi-Paradigm Design for C++, Addison-Wesley, Reading, Mass., 1999.
5. Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns*, Addison Wesley, Reading, MA, 1995.
6. David Garlan, Robert Allen and John Ockerbloom, "Architectural Mismatch: Why Reuse Is So Hard," *IEEE Software*, vol. 12, no. 6, Nov. 1995, pp. 17-26.
7. James Gosling, Frank Yelling and The Java Team, *The Java Application Programming Interface Volume 2*, Addison Wesley, 1996.
8. Gregor Kiczales, Jim des Rivières, Daniel G. Bobrow, The Art of the Metaobject Protocol, MIT Press, 1991.
9. Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier and John Irwin, "Aspect-Oriented Programming," *Proceedings ECOOP'97*, Mehmet Akşit and Satoshi Matsuoka (Ed.), LNCS 1241, Springer-Verlag, Jyvaskyla, Finland, June 1997, pp. 220-242.
10. Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Møller-Pedersen and Kristen Nygaard, "The BETA Programming Language," *Research Directions in Object-Oriented Programming*, B. Shriver and P. Wegner (Ed.), MIT Press, Cambridge, Mass., 1987, pp. 7-48.
11. Markus Lumpe, Jean-Guy Schneider and Oscar Nierstrasz, "Using Metaobjects to Model Concurrent Objects with PICT," *Proceedings of Langages et Modèles à Objets*, Leysin, October 1996, pp. 1-12.
12. *Markus Lumpe, Franz Achermann and Oscar Nierstrasz, "A Formal Language for Composition,"* Foundations of Component Based System, Gary Leavens and Murali Sitaraman (Ed.), pp. 69-90, Cambridge University Press., 2000.
13. Markus Lumpe, "A Pi-Calculus Based Approach to Software Composition," Ph.D. thesis, University of Bern, Institute of Computer Science and Applied Mathematics, January 1999.
14. Mark Lutz, *Programming Python*, O'Reilly, 1996.
15. Robin Milner, "The Polyadic pi Calculus: a tutorial," ECS-LFCS-91-180, Computer Science Dept., University of Edinburgh, Oct. 1991.
16. Michael Morrison, *Presenting Java Beans*, Sams net, 1997.
17. Rocco de Nicola, Gian Luigi Ferrari and R. Pugliese, "Klaim: a Kernel Language for Agents Interaction and Mobility ," IEEE Transactions on Software Engineering (Special Issue on Mobility and Network Aware Computing), Catalin Roman and Ghezzi (Ed.), 1998 .
18. Oscar Nierstrasz, Dennis Tsichritzis, Vicki de Mey and Marc Stadelmann, "Objects + Scripts = Applications," Proceedings, Esprit 1991 Conference, Kluwer Academic Publishers, Dordrecht, NL, 1991, pp. 534-552.
19. Michael Papathomas, "ATOM: An Active object model for enhancing reuse in the development of concurrent software," RR 963-I-LSR-2, IMAG-LSR, Grenoble-France, November 1996.
20. Benjamin C. Pierce and David N. Turner, "Pict: A Programming Language based on the Pi-Calculus," Technical Report, no. CSCI 476, Computer Science Department, Indiana University, March 1997.
21. John H. Reppy, "CML: A Higher-Order Concurrent Language," *ACM SIGPLAN '91 Conference on Programming Language Design and Implementation, SIGPLAN Notices*, vol. 26, no. 6, Toronto, June 26-28, 1991, pp. 293-305.

22. Davide Sangiorgi, "An interpretation of Typed Objects into Typed Pi-calculus," RR 3000, INRIA Sophia-Antipolis, September 1996.
23. Jean-Guy Schneider and Markus Lumpe, "Synchronizing Concurrent Objects in the Pi-Calculus," *Proceedings of Langages et Modèles à Objets '97*, Roland Ducournau and Serge Garlatti (Ed.), Hermes, Roscoff, October 1997, pp. 61-76.
24. Jean-Guy Schneider and Oscar Nierstrasz, "Components, Scripts and Glue," *Software Architectures — Advances and Applications*, Leonor Barroca, Jon Hall and Patrick Hall (Ed.), Springer, 1999, pp. 13-25.
25. Jean-Guy Schneider, "Components, Scripts, and Glue: A conceptual framework for software composition," Ph.D. thesis, University of Bern, Institute of Computer Science and Applied Mathematics, October 1999.
26. Mary Shaw and David Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice-Hall, 1996.
27. Clemens A. Szyperski, *Component Software*, Addison-Wesley, 1998.
28. Patrick Varone, "Implementation of 'Generic Synchronization Policies' in Pict," Technical Report, no. IAM-96-005, University of Bern, Institute of Computer Science and Applied Mathematics, February 1996.
29. David Walker, "Objects in the pi-calculus," *Information and Computing*, vol. 116, no. 2, 1995, pp. 253-271.
30. Larry Wall and Randal L. Schwartz, *Programming Perl 2nd Edition*, O'Reilly & Associates, Inc., 1990.
31. Aaron Watters, Guido van Rossum and James C. Ahlstrom, *Internet Programming with Python*, M&T Books, 1996.

PICCOLA SYNTAX

Form ::=	*'**dynamic**'* | *'**root**'* | *Label* | *Literal*	
	'\' Abstraction	anonymous Abstraction
	Form '.' *Label*	Projection
	Form '(' *Expressions* ')'	Invocation
	Form op *Form*	Infix Invocation
	op Form	Prefix Invocation
	'(' Expressions ')'	
Abstraction ::=	Pattern { '=' | ':' } *Expression*	
Pattern ::=	*'('* [Label] ')' [*Pattern*]	
Expression ::=	[Expressions ','] '**return**' *Form*	local declarations
	Expressions	
Expressions ::=	Statement [',' *Expressions*]	
	Binding [',' *Expressions*]	
Statement ::=	*'**root**'* '=' *Form*	change root context
	*'**dynamic**'* '=' *Form*	*change dynamic context*
Binding ::=	[*'**def**'*] *Label Abstraction*	define service
	[*'**def**'*] *Label* '=' *Form*	assign form
	Label ':' *Form*	define service without arguments
	Form	evaluate Form / add Bindings

Chapter 10

MULTI-DIMENSIONAL SEPARATION OF CONCERNS AND THE HYPERSPACE APPROACH

Harold Ossher and Peri Tarr
IBM T.J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598,
{ossher,tarr}@watson.ibm.com

Keywords: Separation of concerns, software decomposition and composition, modularization, evolution, traceability, limited impact of change.

Abstract: Separation of concerns is at the core of software engineering, and has been for decades. This has led to the invention of many interesting, and effective, modularization approaches. Yet many of the problems it is supposed to alleviate are still with us, including dangerous and expensive invasive change, and obstacles to reuse and component integration. A key reason is that one needs different decompositions according to different concerns at different times, but most languages and modularization approaches support only one "dominant" kind of modularization (e.g., by class in object-oriented languages). Once a system has been decomposed, extensive refactoring and reengineering are needed to remodularize it.

Multi-dimensional separation of concerns allows simultaneous separation according to multiple, arbitrary kinds (dimensions) of concerns, with *on-demand remodularization*. Concerns can overlap and interact. This paper discusses multi-dimensional separation of concerns in general, our particular approach to providing it, called *hyperspaces*, and our support for hyperspaces in Java™, called Hyper/J™.

1. INTRODUCTION

Separation of concerns [20] is at the core of software engineering. In its most general form, it refers to the ability to identify, encapsulate, and manipulate only those parts of software that are relevant to a particular

concept, goal, or purpose. Concerns are the primary motivation for organizing and decomposing software into manageable and comprehensible parts. Many different *kinds* of concerns may be relevant to different developers in different roles, or at different stages of the software lifecycle. For example, the prevalent kind of concern in object-oriented programming is *data* or *class;* each concern in this dimension is a data type defined and encapsulated by a class. *Features* [26], like printing, persistence, and display capabilities, are also common concerns, as are *aspects* [13], like concurrency control and distribution, *roles* [2], *viewpoints* [16], variants, and configurations. We refer to a *kind* of concern, like class or feature, as a *dimension* of concern. Separation of concerns involves decomposition of software according to one or more dimensions of concern. Achieving a "clean" separation of concerns has been hypothesized to reduce software complexity and improve comprehensibility; promote traceability within and across artifacts and throughout the software lifecycle; limit the impact of change, facilitating evolution and non-invasive adaptation and customization; facilitate reuse; and simplify component integration.

These goals, laudable and important as they are, have not yet been achieved in practice. This is primarily because the set of relevant concerns varies over time and is context-sensitive—different development activities, stages of the software lifecycle, developers, and roles often involve concerns of dramatically different kinds and, hence, multiple dimensions. Separation along one dimension of concern may promote some goals and activities, while impeding others; thus, *any* criterion for decomposition and integration will be appropriate for some contexts and requirements, but not for all. For example, the data decomposition in object-oriented systems greatly facilitates evolution of data structure details, because they are encapsulated within single (or a few closely related) classes, but it impedes addition or evolution of features, because they typically include methods and instance variables in multiple classes. Further, multiple dimensions of concern may be relevant *simultaneously*, and they may overlap and interact, as features and classes do. Thus, modularization according to different dimensions of concern is needed for different purposes: sometimes by class, sometimes by feature, sometimes by viewpoint, aspect, role, or other criterion.

These considerations imply that developers must be able to identify, encapsulate, modularize, and manipulate multiple dimensions of concern simultaneously, and to introduce new concerns and dimensions at any point during the software lifecycle, without suffering the effects of invasive modification and rearchitecture. Modern languages and methodologies, however, suffer from a problem we have termed the "tyranny of the dominant decomposition" [25]: they permit the separation and encapsulation of only one kind of concern at a time. Examples of tyrant decompositions are

classes (in object-oriented languages), functions (in functional languages), and rules (in rule-based systems). It is, therefore, impossible to encapsulate and manipulate, for example, features in the object-oriented paradigm, or objects in rule-based systems. Thus, it is impossible to obtain the benefits of different decomposition dimensions throughout the software lifecycle. Developers of an artifact are forced to commit to one, dominant dimension early in the development of that artifact, and changing this decision can have catastrophic consequences for the existing artifact. What is more, artifact languages often constrain the choice of dominant dimension (e.g., it must be *class* in object-oriented software), and different artifacts, such as requirements and design documents, might therefore be forced to use different decompositions, obscuring the relationships between them. We believe the tyranny of the dominant decomposition is the single most significant cause of the failure, to date, to achieve many of the expected benefits of separation of concerns.

We use the term *multi-dimensional separation of concerns* to denote separation of concerns involving:

- Multiple, arbitrary dimensions of concern.
- Separation along these dimensions *simultaneously*.
- The ability to handle new concerns, and new dimensions of concern, *dynamically,* as they arise throughout the software lifecycle.
- Overlapping and interacting concerns; it is appealing to think of many concerns as independent or "orthogonal," but they rarely are in practice.

Full support for multi-dimensional separation of concerns opens the door to *on-demand remodularization,* allowing a developer to choose at any time the best modularization, based on any or all of the concerns, for the development task at hand.

Multi-dimensional separation of concerns represents a set of very ambitious goals. They apply irrespective of software development language or paradigm. No existing mechanism fully satisfies them, and much research remains to be done in pursuit of these goals. We believe that it is necessary to achieve them in order to overcome the problems associated with the tyranny of the dominant decomposition.

The remainder of this paper is organized as follows. Section 2 motivates the need for multi-dimensional separation of concerns by exploring an evolutionary scenario for a simple software system. Section 3 describes multi-dimensional separation of concerns. Section 4 introduces our approach, *hyperspaces*, and Section 5 describes Hyper/J™, our tool support for Java™, and illustrates its use on the evolutionary scenario. Section 6 discusses related work, and Section 7 conclusions and future work.

2. BACKGROUND AND MOTIVATION

To illustrate some of the serious and ubiquitous problems in software engineering that motivate our work, we begin by describing a running example involving the construction and evolution of a simple software engineering environment (SEE), first introduced in [25]. The SEE facilitates the development of fairly simple programs that consist solely of expressions. Expression programs constructed using the SEE are represented using abstract syntax trees (ASTs), as illustrated in *Figure 3*. This environment has a straightforward and commonly used architecture, also shown in *Figure 3*, in which a collection of tools operates on a shared data structure—the AST. Though the example is, of necessity, small and simple, it is typical of a broad class of real systems that involve multiple tools or applications manipulating wholly or partially shared domain models.

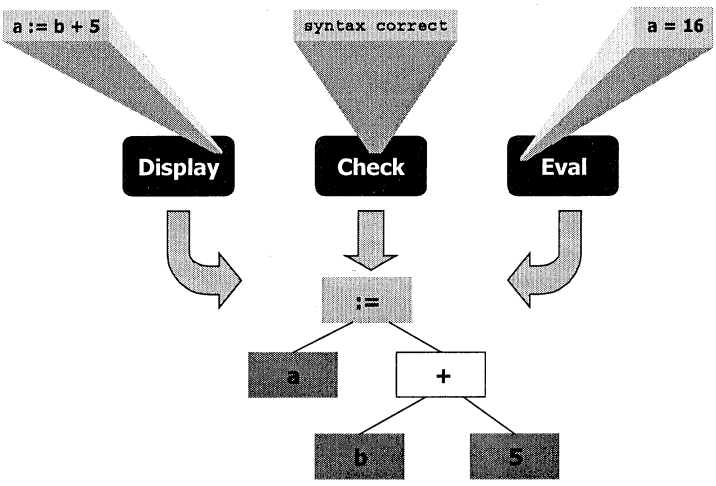


Figure 3: Tools and shared AST in the expression SEE

2.1 The Scenario: A Matter of Concern in an SEE

The running example involves the initial creation of the SEE, and a series of evolutionary changes to it. We assume a simplified initial software development process, consisting of informal requirements specification in natural language, design in UML [22], and implementation in Java™ [7]. The initial requirements specification is straightforward:

> The SEE supports the creation and manipulation of expression programs. It contains a set of tools that share a common representation of expressions. The set of tools should include the following:

- **Evaluation tool**: Determines the result of evaluating an expression and displays it.
- **Display tool**: Depicts an expression program textually to a default display device.
- **Check tool**: Checks an expression program for syntactic and semantic correctness.

A straightforward partial UML design for the SEE is shown in *Figure 4*. This design uses a standard, object-oriented approach, in which a class is defined to represent each kind of expression AST node. Each class contains constructor, accessor and modifier methods, plus methods eval(), display(), and check(), which realize the required tools in a standard, object-oriented manner. The code is structured similarly.

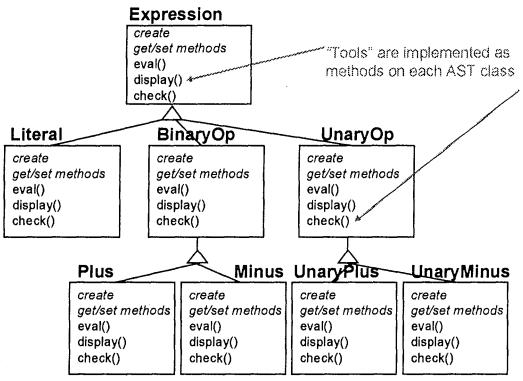


Figure 4: Partial UML design for the expression SE

Even this simple example demonstrates several different kinds (dimensions) of concerns. These include:

Classes (or Objects): Each of the classes in the design and code represents one class concern.

Features: Particularly from the statement of requirements, we can decompose the software into four coherent features: the "kernel" AST, which includes the actual representation of expressions independently of any of the SEE tools; the display feature; the check feature; and the evaluation feature. Note that each feature includes the corresponding requirement specification, design elements, code, and test cases, since these all pertain to addressing that feature concern in the system.

Artifacts: Traditionally, different stages of the software lifecycle have produced different kinds of software artifacts. Some common ones are requirements specifications, designs, code, and test plans.

As noted earlier, we refer to these different kinds of concerns as *dimensions of concern*. Informally, a dimension of concern is simply an approach to decomposing, organizing, and structuring software according to concerns of a particular kind. Note that, despite the clear presence of these different dimensions of concern, only a subset of them can be identified and encapsulated explicitly in the languages used in this example: artifacts, features within the requirements artifact, objects within the design and code artifacts. After using the resulting SEE, the clients request some changes:

- It should be possible to have versions of the SEE that include subsets of the tools and capabilities.
- It should be possible to impose, optionally, checks for conformance to one or more programming styles.
- It should be possible to log, selectively, the execution of the SEE.

This set of modifications suggests the following set of concerns:

- **Configurations**: The first new requirement—to permit different variants of the SEE with different tool configurations—is essentially a request to be able to "mix and match" tools in the SEE. Thus, we can think of the SEE as representing a *family* of software [21], where each member of the family contains some combination of tools.
- **Feature**: Style checking is a new concern in the feature dimension.
- **Logging**: Logging is not the same kind of "feature" as the SEE tools—it is not a coherent tool itself, and it may (optionally) affect some or all of the features during any execution of the SEE.
- **Design patterns**: While the initial version of the software was simple enough not to require any design patterns [6], some of the new requirements present opportunities to benefit from the extra flexibility that design patterns offer. For example, the logging capability could be modeled readily using Observer. From the perspective of comprehensibility, it may be beneficial to look at software in terms of the design patterns from which it is architected [12].

Even without deep analysis, it is clear that making the changes to satisfy these rather simple requirements is by no means a simple matter with standard object-oriented technology. Allowing selection of features and

addition of optional style-checking requires substantial reengineering, probably to introduce design patterns, like Visitor, that provide the needed flexibility. Support for logging requires invasive changes to every method to be logged, such as to perform the logging directly or to participate in Observer design patterns. More detailed analysis of a similar example appeared in [25].

2.2 The Tyranny of the Dominant Decomposition

The primary reason for these difficulties is that the new requirements deal with concerns that are not encapsulated in the original SEE. We can draw some important conclusions from even this simple scenario.

Different decompositions have different properties, both good and bad: Each dimension of concern promotes different subsets of the key software engineering properties noted earlier and, perhaps most importantly, facilitates different kinds of change. For example, objects are units of data abstraction; as such, they aid in isolating users from the details of data representation and implementation, thereby localizing the effects of representation changes. On the other hand, by-class decomposition results in two negative phenomena with respect to features: *scattering* and *tangling*. Features are *scattered* across multiple classes—e.g., each class in the AST design and implementation has its own display() method, the key method in the Display feature. The display method within a class is not isolated: it co-exists—is *tangled*—with methods supporting other features within the same class. Scattering and tangling complicate understanding of how particular functions or features are realized in the software. Further, any change to a function or feature, like Display, has high impact, because scattering implies that it entails modifications to some or all of the classes that define display() methods, and tangling implies that some of the modifications might inadvertently affect other features. Each of the dimensions of concern has both positive and negative software engineering characteristics; there is no "best" dimension for all purposes, which is one reason why different artifact languages are used for different purposes.

Different dimensions are useful for different reasons, at different times: This example demonstrates clearly two crucial points. First, the set of dimensions of concern, and the set of concerns within those dimensions, vary over time. Note, for example, that the design patterns, configurations, and logging dimensions were not relevant to the initial software—they only became relevant with the introduction of new requirements. Second, the fact that each dimension of concern provides only a subset of desirable software

engineering benefits means that developers will find different dimensions to be more or less useful, depending on the developer, his/her role, the stage of development, and the particular goals he/she wishes to accomplish. Thus, for example, adding the new style-check feature would be simple and additive if the software were decomposed by feature, but because features could not be encapsulated, the feature had to be retrofitted into the object dimension. These concerns do not match—they *cut across* [24, 13] each other—so the modification is conceptually difficult, invasive, high-impact, and costly.

Anticipation causes ulcers: Deeply ingrained within software engineering is the notion of anticipating and designing for the "most likely" kinds of changes, towards the goal of limiting the impact of future evolution. Thus, for example, one could argue that a good developer would have anticipated the need for new features, like style checking, and might have designed the software with Visitors from the start, which would have facilitated the introduction of style checking. This is true, and we believe in anticipating and planning for changes whenever possible. Anticipation is not, however, a panacea for evolution. It clearly is not possible to anticipate all major evolutionary directions. Further, even if it were possible, building in evolutionary flexibility always comes at a price: it increases development cost, increases software complexity, reduces performance, or often all of the above. This observation holds for identifying dimensions of concern as well—even if it were possible to identify and encapsulate all possible dimensions, the resulting software would probably be even more complex and unwieldy than it is when only a small number of "dominant" dimensions are encapsulated.

Multiple dimensions, simultaneously: Artifact formalisms (such as programming languages, design notations like UML, etc.) in general provide only one (or a small number of) means of decomposing software—that is, they support only one dimension of concern. In fact, different artifacts for the same software may be written in languages with different "dominant" dimensions, leading to conceptual mismatches between artifacts and to poor traceability, which further complicates evolution. For example, requirements are often specified by function or by feature, as they were in this example—this is how the customers who specify the requirements think of the software—while object-oriented designs and code are decomposed using classes. Thus, developers must continuously translate between different expressions of the same concepts across different formalisms. Unless an artifact language specifically supports a given dimension, it is not possible for developers to identify, separate, and encapsulate concerns along that dimension in that particular artifact. And, as we have seen, if some kinds of

concerns cannot be identified, encapsulated, and manipulated as first-class entities, the software engineering benefits that they might provide cannot be achieved. The "tyranny of the dominant decomposition" becomes oppressive whenever the concerns a developer has, at some point during the lifecycle, do not match any of the ones that have been, or can be, encapsulated. Its symptoms are the kinds of scattering, tangling, and cascaded, high-impact changes that this scenario demonstrates.

3. BREAKING THE TYRANNY: MULTI-DIMENSIONAL SEPARATION OF CONCERNS

The observations in the previous section lead to some important requirements on separation of concerns mechanisms. We use the term *multi-dimensional separation of concerns* to denote separation of concerns mechanisms that satisfy these requirements:

It is necessary for developers to be able to identify and encapsulate *any* kinds, or dimensions, of concern, *simultaneously*. Further, all dimensions must be created equal—there must not be "tyrant" dimensions that preclude decomposition along other dimensions.

Developers must be able to identify new concerns, and new dimensions of concern, *incrementally*, at any time during the course of the software lifecycle. For example, it must be possible to identify only some dimensions, or some of the concerns in a given dimension when the dimension is first introduced, and then identify or introduce others as they are needed, without having to rearchitect the software or make invasive modifications.

Developers must not be required to know about, or pay attention to, any concerns, or dimensions of concern, that do not affect their particular activities. One key purpose of separation of concerns is to reduce the amount of complexity a developer must deal with. Forcing all developers to know about all concerns would, instead, increase this complexity.

It must be possible to represent and manage *overlapping* and *interacting* concerns. As noted earlier, while independent or "orthogonal" concerns have particularly pleasing properties, overlapping and interacting concerns are at least as common in the real world. In representing such concerns, it must also be possible to identify the points of interaction and maintain the appropriate relationships across these concerns as they evolve.

Separation of concerns is clearly of limited use if the concerns that have been separated cannot be integrated together; as Jackson notes, "having divided to conquer, we must reunite to rule" [11]. Thus, any separation of concerns mechanism must also include powerful *integration* mechanisms, to permit the integration of separate concerns.

An important additional goal, though not, in our opinion, a defining characteristic of multi-dimensional separation of concerns, is the ability to impose new decompositions on existing software (i.e., decompose it into concerns along a new dimension), without explicit refactoring, reengineering, or other invasive change. We call this capability *on-demand remodularization.* It allows a developer to choose, at any time, the best modularization for the development task at hand, without perturbing existing ones. In addition to reducing impact of change substantially, this feature opens the door to non-invasive system refactoring and reengineering.

There are potentially many ways to achieve multi-dimensional separation of concerns. As will be discussed in Section 6, there are a variety of modern mechanisms that break the tyranny to at least some extent. The rest of this paper describes our approach, called *hyperspaces*. The goals listed above are extremely challenging, however, and much research remains, for us and for others, before they are fully achieved.

4. THE HYPERSPACE APPROACH

We have developed a particular approach to multi-dimensional separation of concerns that we refer to as *hyperspaces*. Hyperspaces permit the explicit *identification* of any concerns of importance, *encapsulation* of those concerns, identification and management of *relationships* among those concerns, and *integration* of concerns. Many of the decisions we made in defining hyperspaces are aimed at achieving limited impact of change and simplified evolution. We deliberately left some detailed decisions open, to allow for variations with different tradeoffs. We describe hyperspaces in this section, and illustrate their use on the expression SEE example in the next section, in the context of Hyper/J™, a tool that supports hyperspaces for Java™.

4.1 Concern Space of Units

We begin by introducing some terminology that applies to separation of concerns approaches in general. Software consists of *artifacts*, which comprise descriptive material in suitable languages. A *unit* is a syntactic construct in such a language. A unit might be, for example, a declaration, statement, state chart, class, interface, requirement specification, or any other coherent entity that can be described in a given language. We distinguish *primitive* units, which are treated as atomic, from *compound units*, which group units together. Thus, for example, a method, instance variable, or

performance requirement might be treated as a primitive unit, while a class, package, or collaboration diagram might be treated as a compound unit.

A *concern space* encompasses all units in some body of software, such as a set of software systems or component libraries, or a product family. For example, a concern space for the expression SEE contains all of the software artifacts described in Section 2 for both the initial system and the extensions.

The job of a concern space is to organize the units in the body of software so as to *separate* all important concerns, to describe various kinds of interrelationships among concerns, and to indicate how software components and systems can be built and integrated from the units that address these concerns. We identify three distinct components to "separation" of concerns: *identification, encapsulation*, and *integration*. Identification is the process of selecting concerns and populating them with the units that pertain to them.[1] Thus, for example, we can identify the "display feature" concern in the expression SEE as comprising the display requirement and all display() methods in the UML design diagrams and the Java™ code. Identification is useful, but to fully realize the benefits of separation of concerns, the concerns must also be *encapsulated* such that they can be manipulated as first-class entities. A Java™ class is an example of an encapsulated concern. The display feature is not an encapsulated concern in Java™, however, as its units are scattered across many Java™ classes. Once concerns have been encapsulated, it must be possible to *integrate* them to create software that addresses multiple concerns. In standard Java™, classes are integrated simply by loading them; a combination of import specifications and the class path determines their relationships. Concerns other than classes and interfaces cannot be integrated in standard Java™.

4.2 Identification of Concerns: The Concern Matrix

A *hyperspace* is a concern space specially structured to support our approach to multi-dimensional separation of concerns. Its first distinguishing characteristic is that its units are organized in a multi-dimensional matrix. Each axis represents a dimension of concern, and each point on an axis a concern in that dimension. This makes explicit all the dimensions of interest, the concerns that belong to each dimension, and which concerns are affected

[1] Note that concern identification can be done either top-down or bottom-up, depending on the stage of the software lifecycle. During design activities, concerns may be selected first, and then units may be developed based on the concerns that were selected. During system evolution, units may already exist when new concerns are identified. In this case, the identification process determines which existing units address the new concerns.

by which units. The coordinates of a unit indicate all the concerns it affects; the structure clarifies that each unit affects exactly one concern in each dimension. Each dimension can thus be viewed as a partition of the set of units: a particular software decomposition. Any single concern within some dimension defines a hyperplane that contains all the units affecting that concern. The matrix structure permits uniform treatment of all kinds of concerns, and it allows developers to navigate or slice through the matrix according to any desired concerns. We believe that the concerns within a dimension, though disjoint, need not be unrelated, and we expect some concern structure (e.g., hierarchies) within dimensions to be valuable [17, 14]. This remains an issue for future research.

Some dimensions of concern naturally partition the concern space. For example, if every unit in a system addresses exactly one feature, then the Feature dimension naturally partitions the units. However, some units in a system may not pertain to any "feature" at all, such as an error-reporting routine in the SEE. To handle this situation, each dimension in a hyperspace has a specially-designated "none" concern, containing units that are not of interest at all from the perspective of that dimension.

4.2.1 Units

Hyperspaces can be used to organize and manipulate units written in any language(s), though, of course, tool support is often language-specific. To date, we have worked only with units at the granularity of declarations (e.g., methods, functions, classes, UML diagrams) rather than lower-level constructs, such as statements or expressions. We believe, however, that hyperspaces can be extended to handle finer-grained units in a disciplined fashion; this remains an issue for future research.

4.2.2 Concern Specifications

Concern specifications in hyperspaces serve to identify the dimensions and their concerns, and to specify the coordinates of each unit within the matrix. A simple approach is to use a *concern mapping* consisting of specifications of the form

```
x: dimension.concern
```

where x is the name of a unit or a collection of units (e.g., a directory or package), or a pattern representing many units or collections of units. Examples of concern mappings for Java™ units are given in Section 5.

In general, concern specifications can be more complex, and can specify the "meaning" of each dimension and concern formally or informally. There are two styles of specification. *Extensional specifications* explicitly

enumerate the units in each concern. *Intensional specifications* specify properties of concerns and units that can be used to determine whether a given unit pertains to a concern. Intensional specifications have the advantage of conveying intent more explicitly, and of being able to accommodate changes to the underlying set of units without manual intervention.

4.3 Encapsulation of Concerns: Hyperslices

The concern matrix identifies concerns and organizes units according to dimensions and concerns. It allows many useful sets of units to be identified based on the concerns they affect, such as all units pertaining to a single concern, or to all of several concerns (areas of overlap), or to one concern but not another. However, the matrix does not, in itself, support encapsulation of concerns: the sets of units cannot simply be treated as modules, without additional mechanism. In hyperspaces, that additional mechanism is *hyperslices*: sets of units that are *declaratively complete*.

Units are typically related in a variety of ways; for example, one function unit may *invoke* another, or it may *define* or *use* a variable declaration unit. When these kinds of interrelationships exist between units in different concerns, high coupling results. To decouple them, hyperslices are defined to be *declaratively complete*: they must *declare* everything to which they refer. For example, a hyperslice must, at minimum, include a declaration for every function that any of its members invokes, and for any variable its members use. The hyperslice need not provide a full *definition* for these declarations—e.g., it may declare a function without providing an implementation. Thus, declarations can be abstract, specifying (partially or fully, formally or informally) the properties upon which the hyperslice relies.

Declarative completeness is important because it eliminates coupling between hyperslices. Instead of one hyperslice referring to another, thereby depending upon the other specific hyperslice, each hyperslice states what it needs by means of the abstract declarations, thereby remaining self-contained. It does, however, require someone to provide full definitions of the abstractly-declared entities to be fully complete, but *any* appropriate hyperslice(s) can provide these, as will be shown in Section 4.5. This approach therefore fosters flexible configuration and reuse of hyperslices, and is crucial to achieving limited impact of change.

For example, suppose a Display hyperslice contains a unit, Plus.display(), which uses a Plus.getOperand() accessor function, defined in a Kernel hyperslice. To make Display declaratively complete, it must be augmented with its own declaration of Plus.getOperand() (without necessarily implementing it). Plus.display() must then refer to this local declaration,

instead of to the accessor function in the Kernel. This eliminates the coupling between Display and Kernel, in favor of the assertion that the new, abstract declaration must eventually be "bound" to a unit in *some* hyperslice that provides a suitable implementation.

Any set of units can be fashioned into a valid hyperslice by *declaration completion*: providing abstract declarations for everything referenced but not declared within the set. To some extent, this process can be performed automatically, using straightforward (though language-specific) analysis. Automatic declaration completion can determine what declarations are needed, and can create valid declarations. Semantic information associated with declarations—formal or informal specifications—is another matter however, and probably requires human intervention. Specifications on declarations, and the extent to which they can be determined automatically by analysis during declaration completion, remain issues for future research.

Since any set of units can become a hyperslice through declaration completion, arbitrary concerns can be encapsulated using hyperslices. Thus, whatever limitations the underlying artifact language(s) has, and whatever the concern, it is always possible to synthesize a hyperslice that contains just those units pertaining to the concern (plus some abstract declarations).

4.4 Relationships among Concerns

Units, concerns and hyperslices do not exist in isolation; they can be interrelated in a number of different ways. For example, the "display feature" and the "expression class" are related in that they *overlap*—they share some of the same units, as the display() method is part of both concerns—so a change to one concern may affect the other. As another example, we might choose to integrate "syntax check" and "style check" hyperslices to produce a "check" feature that performs both syntax and style checks. In this case, these two hyperslices would be related by one or more *integration* relationships that indicate how they are to be combined.

We can identify two distinct classes of relationships: *context-insensitive* and *context-sensitive*. "Overlap" is an example of a context-insensitive relationship—the "display feature" and "expression class" are always related this way, as long as they share units in common. Integration relationships exemplify context-sensitive relationships—the "syntax check" and "style check" concerns only have this relationship if they are being integrated in some context (e.g., to create a check tool), but the relationship is not inherent in their definition. Other common kinds of concern relationships are "generalizes," "subsumes," and "precludes." Hyperspaces permit the identification and representation of both context-insensitive and context-

sensitive relationships, and their use in analysis (e.g., impact of change) and integration.

4.5 Integration of Concerns: Hypermodules

Hyperslices are building blocks; they can be integrated to form larger building blocks and, eventually, complete systems. For example, to create a working SEE containing the Display hyperslice discussed above, Display must be integrated with some other hyperslice that provides a unit that can be bound to the new, abstract declaration of Plus.getOperand(), to provide an implementation. We refer to this kind of "binding" relationship between units as *correspondence*. Correspondence is a context-sensitive relationship. It occurs within the context of the integration of a particular software component or system—the same declaration unit may be associated, for example, with different implementation units in different systems. In a hyperspace, this integration context is a *hypermodule.*

A *hypermodule* comprises a set of hyperslices being integrated and a set of *integration relationships*, which specify how the hyperslices relate to one another, and how they should be integrated. Correspondence is an important integration relationship, indicating which specific units within the different hyperslices are to be integrated with one another. However, additional details are often needed to specify just how the integration is to occur. For example, if two methods correspond, should one override the other in the integrated system, or are they both to be executed? If both, in what order, and how should the return value be computed? If the types of their parameters do not match, what transformations are needed to reconcile them? In the example above, it is sufficient to integrate the corresponding declarations of Plus.getOperand() in Display and Kernel, which results in the Kernel implementation being called by Plus.display() at run time.

Conceptually, and often in practice through use of a *compositor* tool, the integration specified by integration relationships can actually be performed to produce a set of integrated units. This set will be declaratively complete, and is therefore a hyperslice. A hypermodule can therefore be thought of as a composite hyperslice, produced by integrating a number of subsidiary hyperslices. This implies that hypermodules can be nested, allowing large systems to be built by successive integration.

Declarative completeness, correspondence, and even the more detailed integration relationships, represent fairly loose forms of binding, which promotes evolvability. Since hyperslices do not depend on each other directly, software artifacts are subject to a *completeness constraint* in which each declaration unit in a system must correspond to compatible definition(s) or implementation(s) in some hyperslice(s). Replacing a definition or

implementation is non-invasive on hyperslices; it merely requires the redefinition of integration relationships. Correspondence thus provides great flexibility and directly supports substitutability, including mix-and-match and plug-and-play. Completeness constraints can be imposed as needed (e.g., on code, to ensure that it can run), but they are not necessary when a hypermodule represents a building block (e.g., a reusable component or framework), whose remaining needs can be satisfied through future integration.

Different types of correspondences can occur, beyond the association between declarations and definitions. For example, a requirements unit may correspond to one or more design units that satisfy it; or a unit implementing the eval() function may correspond to a unit that encapsulates code to check for a previously cached result before evaluating. Correspondence and other relationships are deliberately left abstract in this model, as their details depend on many factors, including the language(s) in which units are written, which constructs are treated as primitive and compound units, the extent of environment support for correspondence, etc. Our intent is to provide an abstract model within which multiple semantics for correspondence and multiple realizations of hyperspaces can be specified. Correspondence relationships in Hyper/J™, described in Section 5, extend the *composition rules* from our earlier work on subject-oriented programming [18].

Clearly, the issue of whether corresponding units are "compatible" (e.g., whether an implementation unit satisfies a declaration unit's requirements, or whether a design unit satisfies a requirement) involves both syntactic and semantic issues. How to characterize and check for such compatibility remains an issue for future research. Even once resolved, however, we expect checking to be semi-automatic in general; ultimately, software engineers must understand enough about corresponding units to determine whether or not they are compatible and how best to integrate them.

Hypermodules can be used to encapsulate many kinds of software artifacts, components, and fragments thereof, and to integrate them in different ways. For example, an entire artifact, like a requirements specification, a design, or code, can be modeled as a hypermodule. A software system as a whole is also a hypermodule, subject to the completeness constraint. A system hypermodule might consist of a hyperslice for each artifact, with composition relationships describing how the artifacts interrelate; they might, for example, indicate how particular design and code units elaborate given requirements units. Alternatively, it might consist of a subsidiary hypermodule for each feature, with integration relationships specifying how the features interact. Each feature

hypermodule, in turn, consists of a hyperslice for each artifact, with integration relationships as above.

5. HYPER/J™: HYPERSPACES FOR JAVA™

We have implemented a tool, called Hyper/J™ [10], which supports hyperspaces. It currently supports one language for defining units—Java™—and we have begun looking at incorporating UML also [4]. In this section, we describe Hyper/J™ and illustrate its use by describing its application to the development and evolution of the expression SEE example. We conclude this section by summarizing how Hyper/J™, and the model of hyperspaces it embodies, overcome the problems described in Section 2 to achieve the software engineering goals noted in Section 1.

5.1 The Tool

Hyper/J™ permits the identification, encapsulation and integration (through composition) of multiple dimensions of concern, and it realizes the model of hyperspaces presented in Section 4. It takes as input a *project specification*, which identifies the units (Java™ code) in a given hyperspace; a *concern mapping*, which describes how the units are organized in the concern matrix; and a *hypermodule specification*, which describes hypermodules and controls composition. We will describe these in more detail below. Hyper/J™ can be used at all stages of the software lifecycle, for initial development as well as for extension or evolution of software initially developed with it or without it.

Hyper/J™ includes visual, WYSIWYG support for specifying and integrating concerns. Though its support is still preliminary at present, Hyper/J™ allows developers to identify and manipulate concerns, to focus in on particular dimensions of concern, and to create hypermodules by trial-and-error integration of concerns. A developer starts creating a hypermodule by choosing a set of concerns and an overall default relationship among those concerns (such as "mergeByName," described below). Hyper/J™ creates valid hyperslices for the concerns by automatic declaration completion, and composes them based on the specified relationship(s). The resulting composed hyperslice is displayed. If it is not as desired, the developer can specify new relationships, and examine the new result. S/he can also modify the original concerns; Hyper/J™ will automatically recompute the relationships (deactivating, but not deleting, any that no longer apply) and create a new composed hyperslice that, in many cases, is

either correct or close to what is desired. This interaction continues until the composed hyperslice meets the developer's requirements.

The development of Hyper/J™ was influenced by some important design goals, intended to foster easy, *incremental* adoption. First, we did not want to require developers to adopt new programming languages, or to use special-purpose compilers or virtual machines. We therefore implemented Hyper/J™ to work on and generate standard Java™ class files. All the support for multi-dimensional separation of concerns occurs *outside* the artifact language, Java™. Second, we wanted Hyper/J™ to provide useful benefits when applied to standard Java™ programs, and additional benefits when applied to programs written with Hyper/J™ in mind. It is therefore able to identify, encapsulate and integrate concerns from standard Java™ programs, without requiring special coding conventions or packaging.

5.2 Developing with (and without) Hyper/J™: The Expression SEE in Hyperspace

To convey a sense of the different ways in which developers can leverage Hyper/J's capabilities throughout the software development lifecycle, we present here part of the development process of the expression SEE. Only small illustrations of code are shown here; the full, runnable code for the SEE example is available at the hyperspace web site [10].

5.2.1 Initial Development, without Hyper/J™

To illustrate incremental adoption of Hyper/J™, we assume that the initial SEE was developed using standard object-oriented design and implementation techniques, without Hyper/J™, to produce the design and code shown in *Figure 4* (Section 2.1). Feature concerns are not identified or encapsulated within this code.

5.2.2 Mix-and-Match of Features (and Developing Product Lines) with Hyper/J™

The first change in the requirements entailed permitting the creation of different versions of the expression SEE, each with different subsets of features. Hyper/J™ can help here in two ways. First, it provides on-demand remodularization—the ability to identify and encapsulate new dimensions of concern at any time, without invasive changes. Thus, developers can introduce the needed feature concerns using Hyper/J™, and then manipulate those concerns as first-class entities. Second, Hyper/J's composition capability permits the *selective* integration of concerns, and hence creation of

variants of the expression SEE that integrate different subsets of the available features, as needed, non-invasively.

To use Hyper/J™ to accomplish this task, a developer performs the following steps:

Create a project specification: Developers create hyperspaces initially by specifying a set of Java™ class files that contain the code units that will populate the hyperspace. This is analogous to creating a project or a repository in an integrated development environment. In the Hyper/J™ GUI, developers can choose to add class files at any time using a browser; when using the batch system, developers write a *project specification,* such as:

```
hyperspace Expression_SEE_Hyperspace
  class com.ibm.hyperJ.ExpressionSEE.*;
  file c:\u\smith\com\ibm\hyperJ\util\Set.class
```

Project specifications can contain wildcards, as shown above, and can use either Java™ fully qualified class names or path names of files.

Hyper/J™ automatically creates one dimension—the *Class File* dimension—and it creates one concern in that dimension for each class file it loads. The contents of those concerns are the units (interfaces, classes, methods, and member variables) in the corresponding class files.

Create concern mappings: To achieve the mix-and-match of features that is desired, the developer must first encapsulate the features as first-class concerns. S/he does this by creating a new dimension—the Feature dimension—and describing how existing units in the hyperspace address concerns in that dimension. To do so, s/he specifies *concern mappings,* such as:

```
package  com.ibm.hyperJ.ExpressionSEE: Feature.Kernel
operation display: Feature.Display
operation check: Feature.Check
operation eval: Feature.Eval
```

The first mapping indicates that, by default, all of the units contained within the Java™ package com.ibm.hyperJ.ExpressionSEE address the Kernel concern in the Feature dimension. Since the Feature dimension does not yet exist, Hyper/J™ will create it (and the Kernel concern) upon processing this concern mapping. The other three mappings indicate that any methods named "display," "check," or "eval" address the Display, Check, or Eval features, respectively. These later concern mappings override the first one, whenever they apply. This illustrates an approach employed throughout

Hyper/J™: specification of a general rule followed by exceptions, to clarify and shorten specifications.

The concern matrix now contains two dimensions: Class File and Feature. Each unit addresses exactly one concern in every dimension. Thus, for example, the method Expression.display() addresses the concern com.ibm.hyperJ.ExpressionSEE.Expression in the Class File dimension, and the Display concern in the Feature dimension.

Create hypermodules: Once the feature concerns have been identified, the developer can create versions of the SEE that contain different sets of features by defining *hypermodules*. A *hypermodule specification* comprises:

- A set of hyperslices, specified in terms of concerns identified in the concern matrix. Examples include individual concerns, and unions, intersections or complements of concerns. Hyper/J™ performs declaration completion automatically to create valid hyperslices.
- Integration relationships among the hyperslices and their units. General rules for determining relationships can be followed by exceptions.

For example, the following hypermodule specification defines a version of the SEE that contains the Kernel, Display, and Check capabilities:

```
hypermodule SEE_With_Display_And_Check
  hyperslices: Feature.Kernel, Feature.Display,
               Feature.Check
  relationships:  mergeByName
```

In this hypermodule, the Kernel, Display, and Check concerns are related by a "mergeByName" integration relationship. The "ByName" indicates that units in the different concerns are considered to correspond if they have the same names (and signatures, where appropriate). The "merge" indicates that corresponding entities are to be combined so as to include all their details; for example, all members in corresponding classes are brought together in the composed class. This integration relationship, and many of the others in Hyper/J™, are based on composition rules defined for subject-oriented programming [18].

The hyperslice that results from composing these concerns contains all the AST classes, but with just Kernel, Display and Check functionality in each. In particular, no eval() methods are present.

When the developer is satisfied with the composed hyperslice, s/he can ask to have code generated. Hyper/J™ generates Java™ class files and composed pseudo-source files for any composed hyperslice. The class files can then be executed (or otherwise used) like any other Java™ classes, and the pseudo-source files can be used as "source" in debugging sessions. In the

example, the composed program represents an executable version of the expression SEE that contains the Kernel, Display, and Check functionality only. Other versions of the environment can be created similarly, by defining new hypermodule specifications and using composition. Note that creation of hypermodules is entirely non-invasive, unlike the retrofitting of design patterns usually needed to achieve a similar result.

This part of the scenario has demonstrated the utility of Hyper/J's on-demand remodularization and integration capabilities on *existing* code. Notice that the feature concerns did not have to be identified or separated during initial development to permit them to be encapsulated. Also notice that each of the concerns is itself a reusable component that can be integrated in different contexts with different other concerns—none of them is coupled with any other. These properties imply powerful support for development and configuration of variations within product lines or families.

5.2.3 The Addition of Style Checking

The expression SEE clients eventually requested an enhancement that permits optional style checking of expression programs. Hyper/J™ allows the new feature to be developed separately from the existing features, and non-invasively. That is, developers can write the code for the new feature as a new, separate Java™ package (or packages), which we call a *hyperslice package*, because it is deliberately written to encapsulate a concern. They will then be able to integrate this package with other concerns as needed. This adds tremendous flexibility to the code architectures that developers can select, and to the range of software development processes they can use.

Figure 5 helps to illustrate this point. It depicts the code in the new package that realizes the style checking feature. Notice that the package contains *solely* the code needed to implement the style checking feature (plus abstract declarations, not shown, for anything "foreign" that is used, such as accessor methods from Kernel). Its class structure is similar to that of the original system (*Figure 4*), but not identical, because style checking only affects some of the Expression classes. This is an important feature of multi-dimensional separation of concerns using Hyper/J™: that different concerns can have different perspectives on, or views of, the domain model under development. These different views can later be reconciled by specifying the appropriate relationships between the concerns.

Once the code in *Figure 5* is complete and has been compiled with any Java™ compiler, the developer can add the resulting class files to the existing hyperspace. This results in the automatic addition of new concerns in the Class File dimension—one for each of the class files in the style

checker package. The developer can then define an additional, simple concern mapping to create a Style Checker feature concern:

```
package com.ibm.hyperJ.StyleChecker.* : Feature.StyleChecker
```

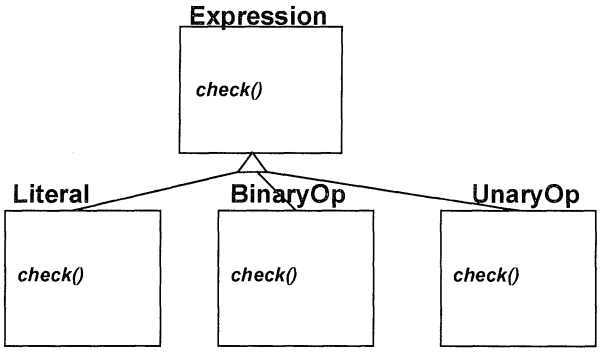


Figure 5: The style checking Hyperslice package

With the style checking feature now identified as a concern, the developer can create variants of the expression SEE that contain style checking or not, as desired, in much the same way as s/he can mix-and-match the other features, described earlier.

The addition of style checking has demonstrated an important feature of Hyper/J™. As shown in Section 5.2.2, developers need not use Hyper/J™ during initial development—they can use it after development—but if they choose to use it during initial development of some part of the system, they can achieve separation of concerns, and code architectures, that would be difficult or impossible to achieve using standard object-oriented techniques. The extra flexibility does not derive from the use of new languages or paradigms—the style checker, for example, was written as a standard package in Java™—but, instead, from the integration (composition) features of Hyper/J™. It has many important advantages and uses, including:

- The ability to treat hyperslice packages as reusable components. When hyperslice packages are used in new contexts, the composition relationships (possibly referring to special-purpose glue code) can include any adaptation that might be necessary (white-box reuse).
- The ability to structure code and design along the same lines as requirements [4], thereby enhancing traceability, by encapsulating the code that realizes a particular requirement in one or more hyperslice packages.

5.2.4 Retrofitting a Design Pattern for Logging

The final change we will explore is the addition of optional logging (or debug tracing) throughout the expression SEE. This modification entails making some or all methods in various classes or features print log messages upon method entry and exit.

Clearly, the logging capability is not specific to the expression SEE—it makes no reference to any expression classes or methods, and the same logging capability could be used in multiple contexts. Thus, the developers determined that they already had a pre-existing, generic, reusable logging component that they could use to satisfy the new end-user requirement. Their reusable component library contains an implementation of the Observer design pattern, along with a particular instantiation of that pattern to implement logging, as shown in *Figure 6*.

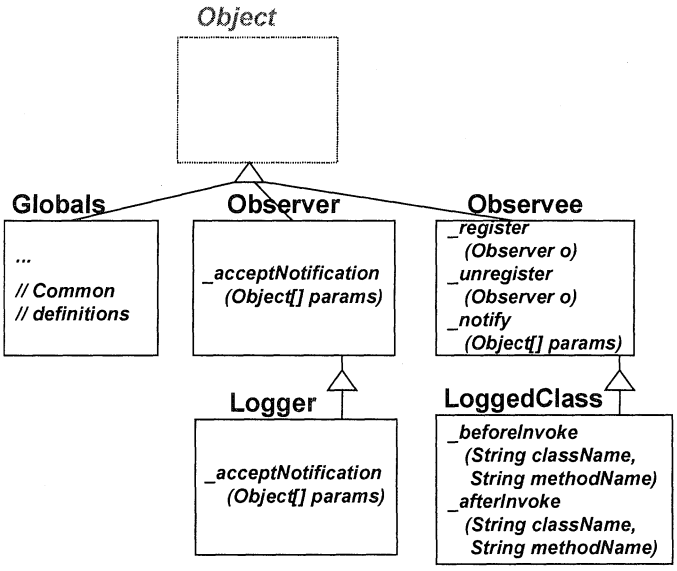


Figure 6: The logging Hyperslice package

In this case, the developers use Hyper/J™ to retrofit the logging capability, which is already encapsulated in a separate hyperslice package (the reusable library), by integrating it into the SEE. Hyper/J™ permits them to make this change additively. The developer simply loads the existing component class files into the hyperspace with the rest of the system, and specifies an appropriate concern mapping to create a new Feature concern, which s/he calls Logging.

To instrument methods in the existing code, the developer defines a hypermodule to integrate the Logging feature with any concerns that s/he wants to be logged. For example, to create a version of the expression SEE that contains the Kernel, Evaluation, and Syntax Check features, with these features logged, s/he might define the following hypermodule:

```
hypermodule SEE_With_Logged_Eval_And_Check
  hyperslices: Feature.Kernel, Feature.Check,
               Feature.Eval, Feature.Logging
  relationships:
    mergeByName;
    merge Feature.Logging.LoggedClass with *;
    bracket ~_* with
      _logEntry(ClassName, MethodName)
      _logExit(ClassName, MethodName)
```

The "merge" relationship expands on the "mergeByName" relationship; it indicates that the LoggedClass unit in the Logging concern from the Feature dimension is to be merged with *all* other class units in the other hyperslices, even though their names differ. Thus, for example, LoggedClass will be merged with the Expression class in Feature.Kernel. Only the same types of units are merged, classes in this case. The "bracket" relationship indicates that each method whose name does not begin with an underscore should be *bracketed* by the methods _logEntry and _logExit. Thus, for example, each display() method in the composed hyperslice will call _logEntry upon entry and _logExit before exit. The parameters passed to these bracketing methods will be the names of the class and method, enabling the logger to identify the method called. The bracket relationship is very useful for circumstances, like this one, where developers need to add behavior to the beginning and/or end of methods.

Notice that this development scenario entailed the integration of generic, reusable components—the Observer design pattern and logging—into an existing system that had not been designed to use them. This is a common problem for developers, and it occurs in many forms, at all stages of software development—for example, integrating a commercial-off-the-shelf database or library component into software during initial development, or retrofitting a design pattern or other component into the software during the course of evolution. Hyper/J™ facilitates a wide range of such integration activities. The same mechanisms can be used both for integration and customization, as this example shows.

We note that the multi-dimensional approach permits integration and customization using *any* concerns, in any dimensions. Thus, for example, while the developers chose to add logging to a subset of *features*, they could

equally well have decided to add it to a subset of *classes*, or to some mix of features and classes. The only difference is in the set of hyperslices specified in the hypermodule. This ability to treat all concerns as equal provides developers the ability to focus their attention on precisely the part of a system that they care about to accomplish their goals.

5.3 Evaluation: Achieving the "ilities"

The development and evolution scenarios just described demonstrate some of the ways in which developers can use Hyper/J™ to facilitate a broad range of common software engineering activities throughout the software lifecycle. We conclude here by evaluating Hyper/J™ briefly with respect to how well it helps developers to achieve the desirable "ility" properties described in Section 1.

Comprehensibility: By their structure, hyperspaces enable software engineers to focus in on any dimensions and concerns of importance that are represented in the hyperspace—they need only examine the hyperplane containing the concern. This facilitates comprehensibility. Further, the ability to define new concerns on demand permits developers to identify concerns on an ongoing basis, as they arise during the course of the software lifecycle. This reduces the initial design and development burden on developers by not forcing them to identify *all* concerns up front that might possibly be important at some point in the software lifecycle. It also improves comprehensibility by permitting developers to identify concerns only when they actually do become important, rather than imposing the cognitive burden of identifying and separating all potentially useful concerns in advance.

The ability to separate, simultaneously, all concerns of importance enables developers to focus on the interactions among concerns and to identify (and encapsulate) new concerns. Hyperspaces make explicit the ways in which concerns affect one another, based on how they intersect and on the interrelationships in which they are involved. These interactions are extremely important for evaluating impact of change on other concerns when working with a particular concern. Concern intersections also often turn out to be useful concerns themselves (e.g., style checking of binary operations, an intersection of feature and class concerns). Hyperspaces provide for the identification and reification of such concerns. Further, the structure of a hyperspace aids the identification of other types of "hidden" concerns. For example, the definition of dimensions precludes situations where two concerns in the same dimension overlap. This property helps identify many

kinds of potential encapsulation errors, ranging from failure to identify concerns, to poor separation of concerns, to coupling of concerns.

Evolvability, limited impact of change, and traceability: Hyperspaces greatly facilitate many aspects of evolution. First, they enable *additive*, rather than *invasive*, extension, customization, and extraction of hyperslices, often even when the changes were not anticipated, as the example shows. This is a significant part of the goal of achieving limited impact of change. Second, the addition of units, concerns, and dimensions to hyperspaces is clearly straightforward, with little impact on existing concerns. Moreover, the process of adding units to a hyperspace (and of defining new concerns), forces developers to determine how the new units (concerns) affect existing concerns (units), even if by only indicating that the new units (concerns) do not affect any existing concerns (units).

Hyperspaces also help to limit the impact of removing concerns, which is typically the most invasive and high-cost evolutionary activity. A common problem in removal is that such changes tend to cascade throughout large parts of a software system. By including the declarative completeness requirement as part of the definition of hyperslices, we have ensured that removal of units, and hence, concerns, can, at worst, affect nothing more than the set of hypermodules in which those units and concerns played explicit roles in relationships. This is because the model eliminates direct dependencies among hyperslices by using declarative completeness. Thus, while removal of a unit from a hyperslice, or a hyperslice itself, from a hyperspace may end up eliminating units with which units in other hypermodules had been *related*, the breaking of these can be followed, *non-invasively*, by the establishment of new relationships to other units in other hyperslices that fulfill the intended semantics of a given concern relationship. The impact of removal can, therefore, be limited to the particular hyperslice it affects and to the relationships within hypermodules. The "dangling relationships" that result from removing units can also be used to identify concerns that might not have been separated appropriately, and to identify other concerns that should also be removed (e.g., when removing a display requirement, we would certainly want to remove any design and code concerns that satisfy the requirement). Thus, the model promotes both traceability and limited impact of change.

6. RELATED WORK

In a prior paper [25], we discussed many modern approaches that introduced novel modularization mechanisms related to multi-dimensional

separation of concerns: subject-oriented programming [8,18], aspect-oriented programming [13], contracts [9], role models [2, 27], adaptive programming [15], Viewpoints [16] and Catalysis [5]. All of these, except Viewpoints, pertain to object-oriented systems, in which the dominant decomposition is by class. Each introduces a mechanism, analogous to hyperslices, to segregate design or code that addresses other, non-class concerns. All these approaches provide some of the benefits of hyperslices, in terms of identification and encapsulation of concerns that are not in the dominant decomposition dimension. Many of these approaches also provide some of the benefits of hypermodules—some degree of flexibility in composition of concerns along some useful dimensions [25]. As such, they all make valuable contributions by satisfying some of the goals of multi-dimensional separation of concerns, but none of them satisfies all.

One key distinguishing characteristic of hyperspaces relative to all other mechanisms known to us is the support for on-demand remodularization: the ability to extract hyperslices to encapsulate concerns that were not separated in the original software artifact. This lowers the entry barrier, greatly facilitates evolution, and opens the door to non-invasive refactoring and reengineering. Other important characteristics of hyperspaces that help to differentiate them from other approaches include:

- Hyperspaces do not restrict the nature or number of dimensions of concern permitted. They allow new concerns and dimensions to be added at any time, and these can apply to existing units, using on-demand remodularization, or to new units.
- Hyperslices are grouping constructs that collect together all software that pertains to a particular concern. Contracts and role models are similar, but not aspects, composition filters or propagation patterns, which are finer-grained, each dealing with part of a concern (e.g., methods that share a weave specification or the filtration needs of a particular object).
- Integration and other kinds of relationships are separate from artifacts. This reduces coupling, which has many benefits. Hyperslices are reusable; different integration relationships can be used in different contexts to specify details of how they are to be reused in those contexts. Separate relationship specifications also permit the hyperspace approach to be applied without changing artifact languages. Composition filters are similar, in that attachment is specified separately from filter code, whereas aspects contain weave specifications and code bundled together.
- Hyperslices are declaratively complete, separately understandable, and reusable. It is possible to understand a hyperslice in isolation, and conceptually possible to specify its semantics. It is also conceptually possible to understand a hypermodule by combining one's understanding

of the component hyperslices and the composition relationships; it is not necessary to create the composed artifact and examine it. Details of semantic specification of hyperslices and composition remain an important area of future research.
- Hyperspaces support primitive units at the granularity of declarations (e.g., functions or members). This is a limitation, but one that simplifies understanding of hyperslices and hypermodules, as well as implementation of composition [19, 25].
- Concerns can span lifecycle phases. Many of the details of making this a practical reality remain to be worked out.

Hyperspaces also relate to, and incorporate results from, other areas of related work. Loose binding is an accepted means of helping to limit the impact of some forms of change. Work in the area of software architecture (e.g., [23, 1]) has identified the need to separate software *components* (like hyperslices) from *connectors* (like relationships). Similarly, earlier work on Precise Interface Control (PIC) [28] identified benefits of representing a particular kind of inter-component interaction: *provides* and *requests*. The declarative completeness requirement and use of separately specified composition relationships are in the spirit of these, and similar, approaches. Barrett et. al. [3] describe a spectrum of mechanisms to achieve connections among components, ranging from tightly to loosely bound, and from early to late binding. We have attempted to choose a point on this spectrum that balances the need to limit the impact of change (by not permitting software components to know about each other) with the need for analyzability (most readily accomplished in the presence of tighter binding).

7. CONCLUSIONS AND FUTURE WORK

A number of important problems in software engineering have resisted general solution, including problems related to the "ilities:" comprehensibility, traceability, and evolvability. We believe that these problems share a common cause: failure to identify and encapsulate, *simultaneously*, all concerns of importance in a software system, and the inability to use different dimensions of concern for different purposes throughout a system's lifetime. This paper presented multi-dimensional separation of concerns as an ambitious set of goals that need to be achieved to address these problems fully. It also presented our approach to achieving them, called *hyperspaces*, and its realization in tool support for Java™, called Hyper/J™.

Hyperspaces permit the identification, encapsulation, and integration of any kinds of concerns in standard software, either during initial system development or in retrospect, as the need arises during the course of evolution. They allow the set of concerns of interest to grow and change. They permit explicit representation of relationships among units, concerns and dimensions, loose coupling among concerns, and on-demand remodularization. They even allow for the representation of concerns that span the software lifecycle—for example, the "display" concern has requirements, design, code, and test module units associated with it. Because developers have the choice of when and how to apply hyperspaces, hyperspaces do not interfere with existing software processes—though developers may choose to modify their processes to take advantage of the extra flexibility hyperspaces give them—and the entry barrier for developers to use them is quite low.

This work is clearly at an early stage, largely unproven in practice as yet. Still, a considerable body of experience and related research now exists to support the claim that multi-dimensional separation of concerns is one of the key software engineering issues today, and hence an important area for research. Many questions remain, such as: What sorts of concerns are important in real software development projects? What are their structure and the nature of their interactions, for concerns both within and across software artifacts? How can one understand, and perhaps specify, encapsulated concerns and their integration? What mechanisms are needed to achieve the goals of multi-dimensional separation of concerns? How do they scale? What tool support is needed? How can the software process be improved to take advantage of these ideas and tools? As these and many other open questions are answered, and tools are built, it will become possible to apply multi-dimensional separation of concerns to real development, and thus to explore its benefits and its limits.

8. REFERENCES

1. R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, July, 1997.
2. E. P. Andersen and T. Reenskaug. System design by composing structures of interacting objects. In *ECOOP '92: European Conference on Object-Oriented Programming*, Springer-Verlag. LNCS, no. 615, pp. 133–152, Utrecht, June/July 1992.
3. D. J. Barrett, et al. A Framework for Event-Based Software Integration. *ACM Transactions on Software Engineering and Methodology*, 5(4):378–421, October 1996.
4. S. Clarke, W. Harrison, H. Ossher and P. Tarr. Subject-oriented design: Towards improved alignment of requirements, design and code. In *Proceedings of the Conference*

on Object-Oriented Programming: Systems, Languages, and Applications, pages 325–339, November, 1999. ACM.

5. D. D'Souza and A. C. Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach.* Addison-Wesley, 1998.
6. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.
7. J. Gosling, et al.. *The Java™ Language Specification.* Addison-Wesley, 1996.
8. W. Harrison and H. Ossher. Subject-oriented programming (a critique of pure objects). In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications*, pages 411–428, September 1993. ACM.
9. I. M. Holland. Specifying reusable components using contracts. In *ECOOP '92: European Conference on Object-Oriented Programming*, Springer-Verlag. LNCS 615, pp. 287–308, Utrecht, June/July 1992.
10. Hyperspace web site, http://www.research.ibm.com/hyperspace.
11. M. Jackson. Some complexities in computer-based systems and their implications for system development. In *Proceedings of the International Conference on Computer Systems and Software Engineering*, pages 344–351, 1990.
12. R. K. Keller, et al. Pattern-Based Reverse-Engineering of Design Components. In *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, pages 226–235, May, 1999.
13. G. Kiczales. Aspect-oriented programming. *In ECOOP '97: European Conference on Object-Oriented Programming*, 1997. Invited presentation.
14. Doug Kimelman, Multidimensional tree-structured spaces for separation of concerns in software development environments. Position paper, OOPSLA '99 Workshop on Multi-Dimensional Separation of Concerns in Object-Oriented Systems, http://www.cs.ubc.ca/~murphy/multid-workshop-oopsla99.
15. M. Mezini and K. Lieberherr. Adaptive plug-and-play components for evolutionary software development. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications*, October, 1998.
16. B. Nuseibeh, J. Kramer, and A. Finkelstein. A framework for expressing the relationships between multiple views in requirements specifications. *IEEE Transactions on Software Engineering*, 20(10):760–773, October, 1994.
17. H. Ossher. A case study in structure specification: A Grid description of scribe. *IEEE Transactions on Software Engineering*, 15(11), 1397–1416, November, 1989.
18. H. Ossher, M. Kaplan, A. Katz, W. Harrison, and V. Kruskal. Specifying subject-oriented composition. *Theory and Practice of Object Systems*, 2(3):179–202, 1996.
19. H. Ossher and P. Tarr, Operation-level composition: A case in (join) point. In *ECOOP '98 Workshop Reader*, pages 406–409, July 1998. Springer Verlag. LNCS 1543.
20. D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.
21. D. L. Parnas, On the Design and Development of Program Families. In *IEEE Transactions on Software Engineering*, 2(1), March 1976.
22. J. Rumbaugh, I. Jacobson, and G. Booch. *Unified Modeling Language Reference Manual.* Addison-Wesley, 1998.
23. M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice Hall, April 1996.
24. P. L. Tarr and L. A. Clarke. PLEIADES: An object management system for software engineering environments. In *Proceedings of the ACM SIGSOFT '93 Symposium on Foundations of Software Engineering*, pages 56—70, December, 1993.

25. P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton, Jr. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 21st International Conference on Software Engineering (ICSE'99),* 107–119, May 1999.
26. C. R. Turner, et al. Feature Engineering. In *Proceedings of the 9th International Workshop on Software Specification and Design*, 162–164, April, 1998.
27. M. VanHilst and D. Notkin. Using roles components to implement collaboration-based designs. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications*, pages 359–369, October 1996. ACM.
28. A. L. Wolf, L. A. Clarke, and J. C. Wileden. The AdaPIC toolset: Supporting interface control and analysis throughout the software development process. *IEEE Transactions on Software Engineering*, 15(3):250–263, March 1989.

Chapter 11

COMPONENT INTEGRATION WITH PLUGGABLE COMPOSITE ADAPTERS

Mira Mezini[*], Linda Seiter[**] and Karl Lieberherr[***]

[*] *Department of Computer Science, Darmstadt University of Technology Wilhelminenstraße 7, D-64283 Darmstadt, Germany. email: mezini@informatik.tu-darmstadt.de*
[**] *Department of Computer Engineering, Santa Clara University, Santa Clara, CA 95053, USA. Email: lseiter@scu.edu*
[***] *College of Computer Science, Northeastern University, Cullinane Hall, 360 Huntington Avenue, Boston, Massachusetts 02115, USA. email: lieber@ccs.neu.edu*

Keywords: Business process integration, separation of concerns, object-oriented frameworks, component-based programming, component integration

Abstract: In this chapter we address object-oriented component integration issues. We argue that traditional framework customization techniques are inappropriate for component-based programming since they lack support for non-invasive, encapsulated, dynamic customization. We propose a new language construct, called a *pluggable composite adapter*, for expressing component gluing. A pluggable composite adapter allows the separation of customization code from component implementation, resulting in better modularity, flexible extensibility, and improved maintenance and understandability. We also discuss alternative realizations of the construct.

1. INTRODUCTION

Component software, i.e., software that is an assembly of individual, independently developed parts, is becoming the predominant architecture. We consider two factors as the driving force behind this development. First, as indicated in [23] component software represents a middle path between the two extremes that predominate traditional software development: (a)

custom-made software, i.e., software that is developed from scratch with only the help of tools/libraries and (b) standard software, i.e., prefabricated complete solutions that can only be parameterized to get close enough to what is needed in a particular scenario [23].

The first approach has the advantage that the resulting software optimally supports the specific needs of the particular customer, i.e. competitive strategic decisions, as well as changes in the business process. However, it has severe cost, along with maintenance and evolution problems that make it practically obsolete. The opposite is true for the second approach. While it is certainly the more effective approach, it results in solutions that totally ignore the needs of particular customers and their strategic decisions. Component software combines the advantages of the traditional approaches while avoiding their problems: most of the individual components can be standard solutions with all the advantages that this brings, while on the other side customer-specific strategic decisions can be individually satisfied by integrating custom-made parts and adapting the standard components during the assembly process.

The second factor in favor of component software is that most of the effort in developing an enterprise application actually goes into the development of business logic that is shared across the application's vertical domain. Feedback from companies that plan to deliver applications based on IBM's SanFrancisco framework - a Java-based collection of components that allows developers to assemble server-side business applications from existing parts - indicates that as much as 80% of their development cost is spent writing and supporting the basic, non-competitive functions that are essentially the same for any application solution offered in a specific domain [21]. In such conditions, the benefits of concentrating the fabrication of the shared functions in a few components and reusing them across domains is obvious. This observation has actually driven the SanFrancisco project - one of the few server-based component frameworks today.

Now, if we consider component-based development to predominate server-based solutions, the architecture of server-side software resulting from this development will have more or less the layered structure in Figure 1. The figure is a slightly modified version of that representing the structure of SanFrancisco-based server systems found in [21], which shows the architecture for Java server-side software. However, the architecture is language independent, provided that server-side component models for component transaction monitors similar to Enterprise JavaBeans [22] are available for these languages.

The interest of our work is in the boundary between the upper two horizontal layers. Software parts in the core business processes layer (object-oriented frameworks or EJB components in a Java setting, e.g., for the

domain of Warehouse Management) are provided by component vendors. They model partial solutions while leaving open application-specific details. Software building blocks in the top layer are developed by application developers preferably independent of the components in the lower layer. In an end-product, the elements from these two layers obviously need to be coded into a single solution.

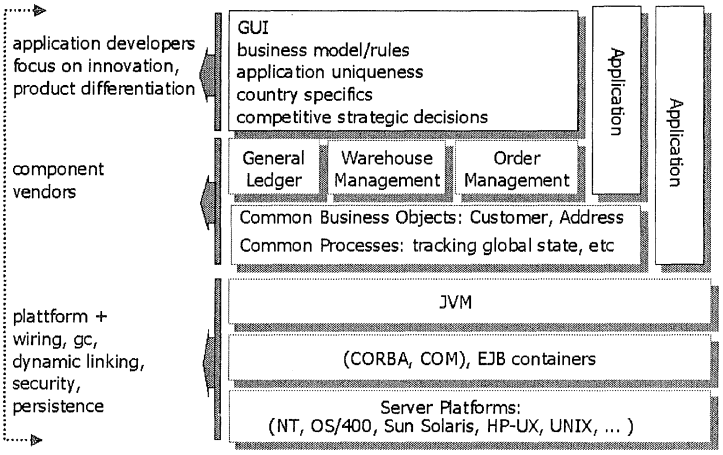


Figure 1: Server-side component software

In order to ensure that full-advantage is taken from the component-based architecture, it is important that the integration technique satisfy the following requirements:

- *The integration process should work with independently developed components*. Pre-fabricated components from the core business processes layer may in general be the product of different vendors. Furthermore, components in the core business processes layer should be independent of the application-specific components found in the upper-most layer. The application developer might have legacy code (e.g., class libraries) that he/she wants to integrate into the end-product. This requirement implies that the integration process should tolerate incompatible interfaces among the components to be integrated.
- *The integration technique should enable flexible (dynamic) reconfiguration of the integrated system*. This implies easy integration of new components into an integrated setting to reflect changes in the business structure, as well as dynamic reconfiguration to enable dynamic adaptation of the system's behavior. Individual components often come in different implementations and it is necessary to dynamically switch

between these different flavors. Furthermore, it makes sense to plug certain components in and out depending on runtime conditions.

Cross domain business logic will most likely be modeled as object-oriented frameworks: business processes in the SanFrancisco project are indeed modeled as application frameworks. Thus, the end-product of component-based application development will typically include several frameworks, as well as class libraries containing either legacy code or application-specific functionality, e.g., encoding the business structure of the particular application. However, traditional framework-based development is based on *reuse by extension* [12]. Frameworks are generally developed to be deployed by newly written application specific code. For instance, the techniques used for building SanFrancisco framework-based applications [21] are based on a mixture of subclassing and aggregation (class- and object-based composition).

We argue that traditional framework-based development does not properly support the requirements posed above for component integration techniques. Similar observations about problems with framework composition are made in [12]. We argue that given the importance of the integration process, object-oriented languages should provide explicit constructs for capturing abstractions in this process. We propose to extend object-oriented languages with an explicit scoping construct for component integration, called a *Pluggable Composite Adapter*. Having a dedicated construct for capturing the component integration apart from the code that models business logic improves the modularity of the end-product. Integration of new components into the system is equivalent to defining a new adapter: the implementation of the existing parts is not affected by the integration. Adapters are first-class objects in this model thus supporting the dynamic reconfiguration of the system. By being themselves components, adapters can be aggregated into more complex building blocks. Furthermore, refinement techniques ´a la inheritance apply. We have developed a prototype implementation of the *Pluggable Composite Adapter* in Java.

The remainder of the chapter is organized as follows. In Section 2, we present a running example to demonstrate the shortcomings of traditional component composition techniques. The *pluggable composite adapter* model is presented in Section 3. Section 4 briefly presents the current prototype realization of the model in Java and outlines possible alternative realizations. Related work, a summary of the paper and areas of future work, are discussed in Section 5.

2. TRADITIONAL OBJECT-ORIENTED COMPOSITION TECHNIQUES

In this section we demonstrate the problems arising from traditional component integration techniques in terms of a running example. A more general discussion of framework composition issues can be found in [12].

Assume we are given a component that encodes the process of price calculation in the domain of order processing[1]. The component is part of a framework for order processing systems. It is intended to be customized by different applications to customer-specific pricing schemes.

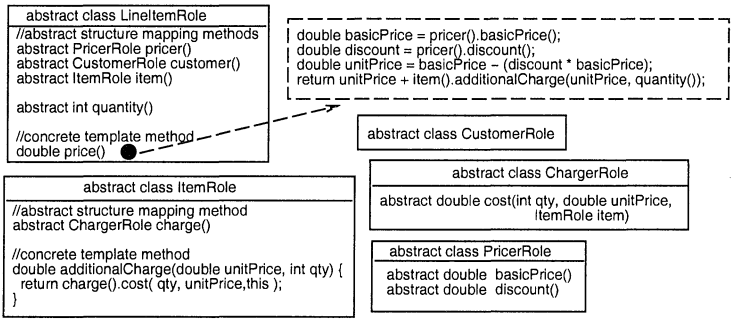


Figure 2: Pricing framework class model

Figure 2 contains the pricing framework component. LineItemRole is responsible for calculating the actual price of a line item purchased by a customer. PricerRole provides price and discount information for the item and customer encapsulated by LineItemRole. ItemRole is responsible for calculating additional charges, defined in ChargerRole. Figure 2 also defines the collaboration required to compute the price of a line item. The price and additionalCharge methods are template methods that define the message and data flow of the collaboration. The primitive operations (basicPrice, discount, etc.) are abstract, to be filled in with an application-specific implementation.

The class diagram modeling the business model of an example application, a product component, is shown in Figure 3. Assume we wish to deploy the pricing component with the product application according to three pricing schemes. Each scheme requires the application component to conform to the framework component in a different way.

– *Regular Pricing*: Each product has a base price that may be discounted based on quantity ordered. Quote plays the LineItemRole, while HWProduct

[1] The definition of this component is first described in [9].

plays the ItemRole. In this pricing scheme HWProduct will also play PricerRole, implementing basicPrice, discount for regular pricing by calling regPrice and regDiscount, respectively. Tax plays the ChargerRole, implementing the cost method. Finally, Customer maintains the customer role.

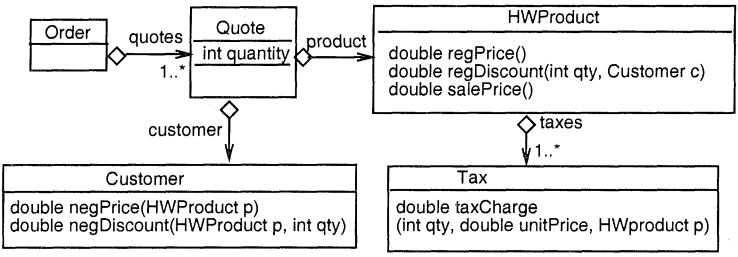


Figure 3: Product application class model

- *Negotiated Pricing*: A customer may have negotiated certain item prices and discounts. Quote plays the LineItemRole. Customer plays the PricerRole, implementing basicPrice, discount for negotiated pricing. Customer also plays the CustomerRole. HWProduct plays the ItemRole. Finally, Tax plays the ChargerRole.
- *Sale Pricing*: Each product has a designated sale price and no discounting is allowed. Quote, HWProduct, Tax and Customer play the same roles as they do with the regular pricing scheme. However, HWProduct will implement basicPrice and discount for sales pricing rather than regular pricing, i.e., by calling salePrice and returning zero respectively.

The traditional framework deployment technique uses static inheritance to model *plays-the-role-of* mappings. For example, the regular pricing scheme would require Quote to be redefined as a subclass of LineItemRole, HWProduct to be redefined as a subclass of both ItemRole and PricerRole, etc. Note that each of the three pricing schemes requires multiple inheritance. In languages that do not support multiple inheritance, e.g., Java, some of the mappings would be established indirectly using a technique such as the adapter design pattern [5]. Framework deployment using static inheritance has three drawbacks. First, it is invasive in that it requires modification of the application classes to encode the customization and the inheritance relationships. Second, it does not encapsulate the multiple roles of the pricing scheme into a single construct, as the roles would be spread out among the various application classes. Third, it is static in that it restricts the product application to a particular pricing implementation at the point of Quote class instantiation.

Accommodating all three pricing schemes and allowing an application to dynamically switch between them by applying design patterns [5] results in the design given in Figure 4. Classes that were present in the original application model are represented in Figure 4 by the non-filled rectangles. The adapter pattern [5] is used in two places to adapt the interfaces of HWProduct and Customer to their PricerRole (the adapter classes HWProduct_PricerRole and Customer_PricerRole, respectively). The strategy pattern is used to model the three pricing variants (regular, negotiated, and sale) of the LineItemRole played by Quote. Each of these classes is parameterized with a different PricerRole adapter (RegPricer, SalePricer, and Customer_PricerRole, respectively). Finally, there will be *facade* classes for the price calculation according to each scheme. Each of these classes would implement the method price(Quote q) by setting the appropriate strategy and adapter objects for the quote argument, and then invoke price() on it. The facades are not shown in Figure 4 due to lack of space.

The figure indicates several shortcomings of traditional object-oriented framework composition techniques. First, deployment of the pricing functionality with the product application requires the modification of the application classes. That is, it is not possible to apply newly acquired business processes to existing objects. Second, the integration of the pricing component results in a proliferation of classes and spurious relations in which the original design gets lost. The clarity of the design is damaged because the code responsible for one customization is not localized in one place, but rather spread around several new adapter, strategy, or facade classes as well as in the original classes. The resulting tangled code is difficult to understand, maintain and extend with new components. This is especially true in a real-life scenario where many business objects and processes can be expected to be involved. To summarize, one could say that the design in Figure 4 suffers from poor modularity.

Recall that it is desirable to non-invasively integrate new components into an application in such a way that the newly integrated functionality can be used with existing objects. This can be realized by advanced design and implementation techniques that combine several simple design patterns. We have presented such a design technique and an elegant implementation of it in [16]. The technique, called the *composite adapter design pattern*, is a generalization of the original adapter pattern [5] that supports the adaptation of collaborative functionality. The original adapter pattern, whose structure is given in Figure 5, solves the problem of adapting the interface of a single pre-defined class (Adaptee in Figure 5) to match the interface of a single class in a framework (Target in Figure 5).

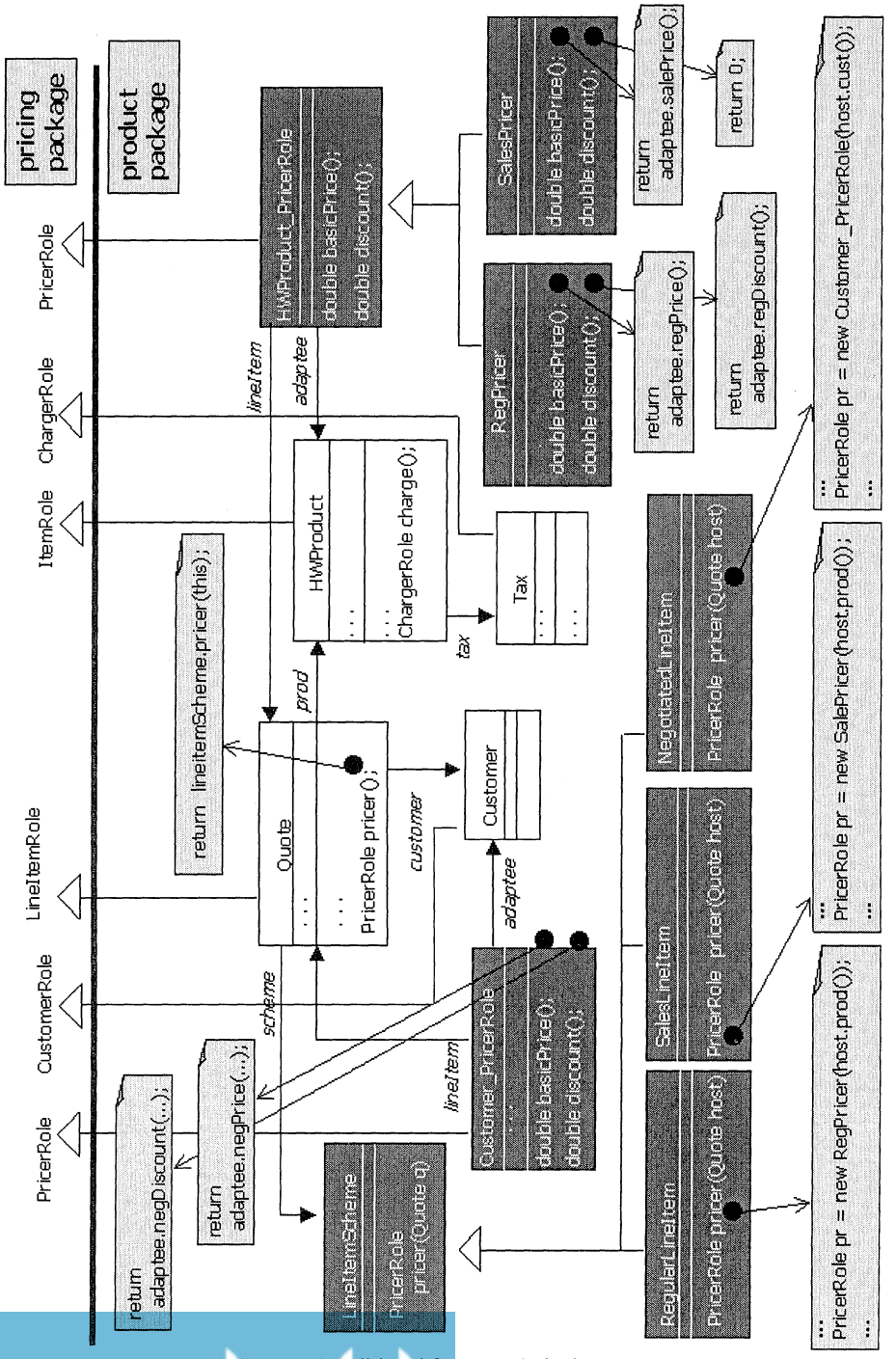


Figure 4: Traditional framework deployment

However, the pattern does not deal with collaborations (composite object adaptation) and the typing issues that arise when one needs to map a set of target classes to a set of adaptee classes. Note that the sample interface used in the structure of the pattern has no return values or parameters. What if the signature of specificRequest was SpecificReturnType specificRequest (SpecificArgType$_1$,...SpecificArgType$_n$), where SpecificReturnType and SpecificArgType$_1$, ...,SpecificArgType$_n$ are types in the domain of the class model that defines Adaptee? These issues are not considered by the original adapter pattern, however they are solved by the composite adapter technique described in [16]. The composite adapter technique provides a means for integrating collaborative functionality and business processes into existing applications.

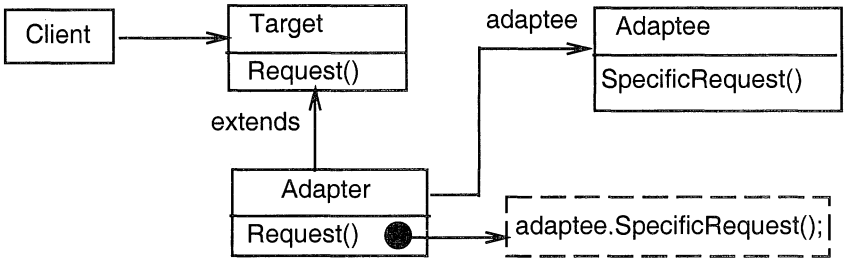


Figure 5: The structure of the original GOF adapter design pattern

Figure 6 shows the composite adapter design pattern technique applied to the integration of sample Java framework and application components. We summarize the technique as follows. Assume we want to integrate the functionality implemented in the Framework package into the application class model given in the Application package. There will be a composite adapter responsible for this integration (App_Frm_CompositeAdapter), which will have a nested class adapter defined for each framework class (RootAdapter, ChildAdapter). Each nested adapter extends a framework class, implementing it in terms of an application class. Thus, each nested adapter maps the interface of an application class (referenced by *adaptee*) to the interface of the framework class that it extends.

The nested adapters RootAdapter and ChildAdapter are essentially adapters in the sense of the original adapter pattern [5]. However, the nested adapters must also take care that application objects (AppRoot, AppChild) that come into the scope of the nested code must be wrapped by the corresponding class adaptation (RootAdapter, ChildAdapter) before being operated on. The class adaptations represent dynamic extensions of the application classes, thus the application class instances should appear to acquire their dynamic types while they are referenced within the composite adapter implementations.

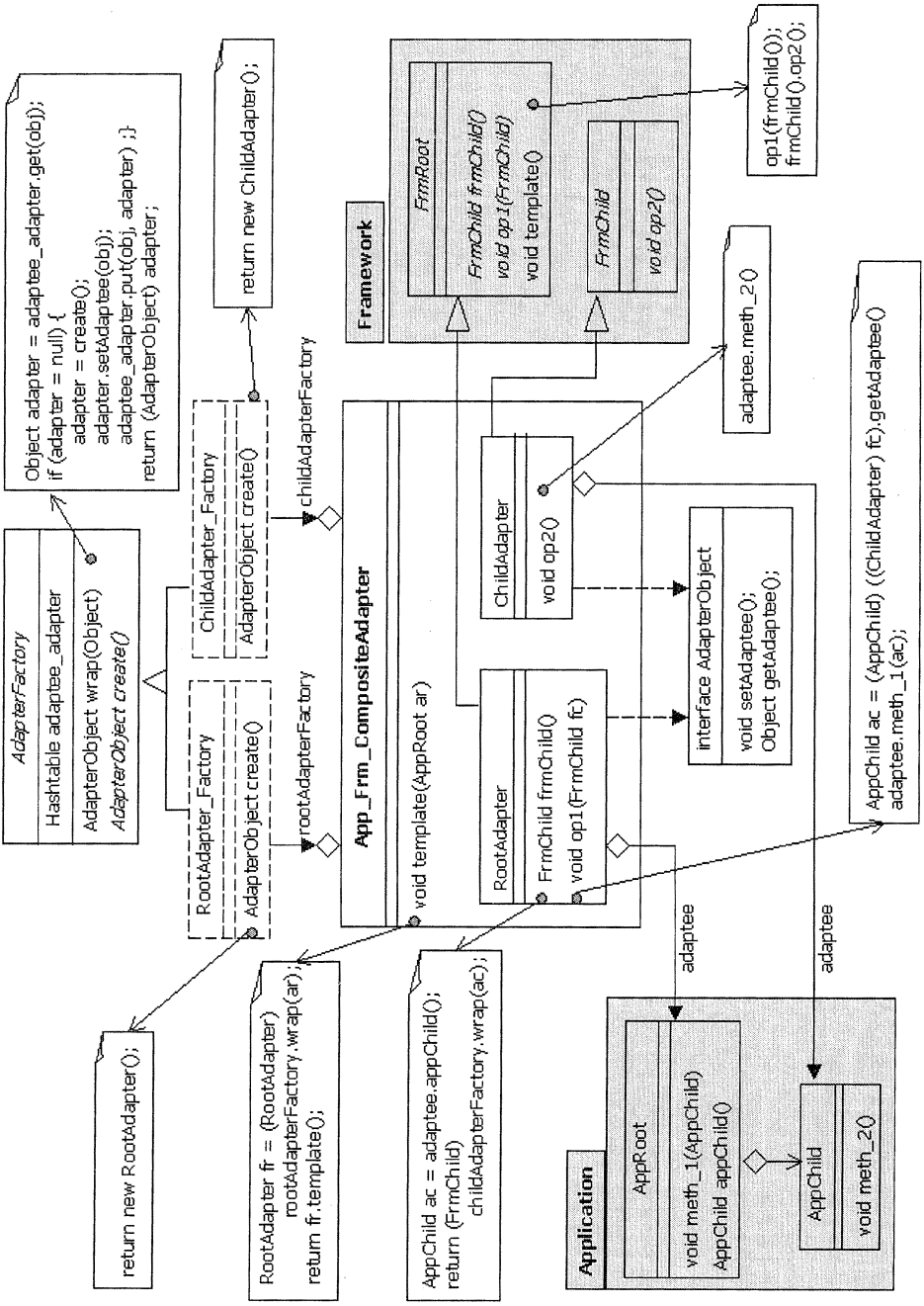


Figure 6: Framework deployment with the composite adapter

The application objects acquire their dynamic types by being wrapped by the nested class adapter objects (see e.g., the implementation of frmChild() in RootAdapter in Figure 6. The creation of class adapter objects is done by factories that memorize the established adapter-adaptee relations, (RootAdapter_Factory and ChildAdapter_Factory shown in the dashed rectangles in Figure 6 to indicate that they are created within App_Frm_CompositeAdapter as anonymous inner classes of the predefined AdapterFactory).

The solution in Figure 6 does indeed satisfy the requirement that the application can be developed independently from the framework[2]. However, the structure of the adapter is complex and the technique is too advanced to expect it to become a common tool in the toolbox of an average programmer. One should not forget that patterns are conventions that need to be followed, as such they are not enforced by the language and cannot simply be assumed by average programmers [8, 2]. A better alternative would be to establish some kind of "plays-the-role-of" wiring relationship between classes in different components independently of the component implementations. It is exactly here that the *Pluggable Composite Adapter* construct comes into play.

3. PLUGGABLE COMPOSITE ADAPTERS

Given the importance of the integration process in component-based development, we argue that it should be supported by dedicated language constructs, and propose to extend object-oriented languages with the *pluggable composite adapter* construct to serve this need.

Before outlining the features of this construct, however, let us first briefly define the term component as it is understood in the context of this paper. As indicated by the discussion so far, we understand a component to be a set of (collaborating) classes that define some (incomplete) functionality. In general, a class model could be captured by some module construct such as a Java package. However, it is preferable for a component to be a closed entity with well defined *expected* and *provided interfaces* [23, 15]. Java packages are not closed entities in the sense that one can change the interface of a package after it is defined by simply implementing new classes and declaring them as members of the package. Examples of closed component constructs are *Adaptive Plug & Play Components* (*AP & PCs*) proposed in [15], and the generalized class model in Java, where classes can be nested

[2] The technique has been originally formulated for framework-application integration, but it can be easily extended to serve the case when two fully-implemented components are to be integrated (replacing the inheritance links by usage links).

into other classes. In this section we will indicate the advantages of having a component be a closed entity. However, to avoid restricting the applicability of the composite adapter construct to non-mainstream models such as *AP & PCs* [15] or making the use of Java inner classes a precondition, in the following we will use the term component as equivalent to package and silently assume that once a package component is defined its interface remains stable.

Having clarified our use of the term component, let us now turn back to the composite adapter construct. In its simplest form, a *pluggable composite adapter* defines how to *extend* a component C_1 with a new concrete collaboration. This use of the *pluggable composite adapter* is also called *component adaptation*. In addition, a *pluggable composite adapter* may define how to glue together two independently developed components C_1 and C_2, where C_2 is an abstract collaboration. The gluing extends C_1 with a concrete realization of the abstract collaboration C_2 in terms of the functionality provided by C_1. We call this use of the *pluggable composite adapter* construct *component gluing*.

A collaboration involves a set of objects that work together to implement a task, where each object plays a particular role. Hence, extending a component with a collaboration will require the adaptation of several classes in the component to the appropriate collaboration roles. New variable, method, and class definitions might be needed in order to realize the component adaptation. Hence, the structure of a *pluggable composite adapter* presented in Figure 7. The nested adapters (i.e., adapter R adapts C_1.B [extends C_2.S] adaptation_body in Figure 7) are called *class adapters* because they specify the adaptation of a single class B of component C_1 to its role R. The enclosing adapter A is called a *composite adapter*, as it nests a set of class adapters. The component C_1 is referred to as the base component (hence the denotation B for classes in this component). In the case of component gluing, the abstract collaboration C_2 that the base component is being adapted to is referred to as the super component (hence the denotation S for its classes).

The attribute *pluggable* reflects the fact that the component adaptation/gluing encoded within a composite adapter may be plugged in and out as needed. Several alternative realizations of the composite adapter construct may differ on whether the process of plugging in/out is dynamic or not. In this section we present the dynamic flavor of the composite adapter, while alternative flavors will be briefly outlined in the following section. "Dynamic" means: C_1 objects are adapted on the fly to play the roles in the collaboration without physically changing C_1's classes. A composite adapter's instances will dynamically lift instances of classes in the base component to the types defined in the nested class adapters (hence, to the

types in the super component in the case of component gluing, since the latter will be supertypes of the nested class adapters). A composite adapter's instances will likewise dynamically lower the previously lifted base instances, restoring them to their original types as necessary. In other words, the composite adapter serves as a "simulation" of the abstract model it defines or extends in terms of the concrete model of component C_1.

adapter A {
    Field_Method_Defs
    Helper_Class_Defs
    { **adapter** R **adapts** C_1.B [**extends** C_2.S] adaptation_body }*
}

Figure 7: The structure of the pluggable composite adapter construct

The *extends* clause is put within brackets in Figure 7 to indicate that it is optional. In the following, we present the main concepts of the *pluggable composite adapter* model by considering first the case when there is no *extends* clause, i.e., the case of dynamic component adaptation in which the adapter directly implements a concrete collaboration. We will subsequently consider dynamic component gluing in which the *extends* clause will be used to support the concrete realization of the abstract collaboration given in the super component.

3.1 Dynamic Component Adaptation

In Figure 7, a class adapter adapter R adapts C_1.B adaptation_body implies an adaptation of the functionality of base component class C_1.B by the code in the adaptation_body. The adaptation_body encodes the delta by which we would have to enhance the definition of class C_1.B to play the given collaboration role R if we used static subclassing, or if we modified the implementation of C_1.B in-place. However, the dynamic version of the *adapts* relation implies neither an in-place modification of C_1.B, nor does it create a new subclass of C_1.B. Via the *adapts* relation we define a "dynamic" extension of C_1.B, meaning that (existing) objects b: C_1.B appear to acquire the extension given in adaptation_body.

The adaptation_body can be thought of as written in terms of three "*self* variables" (with the following precedence): *R.this* (the role environment), *B.this* (the base or adaptee environment), and *A.this* (the composite adapter environment). All three self variables provide environments for binding variable and method definitions. That is, (a) the definitions in the nested class adapter R, (b) the definitions in base component class C_1.B, and (c) the definitions in the enclosing composite adapter A (Field Method Defs, Helper Class Defs) are within the scope of the adaptation body.

For illustration, Figure 8 shows an example of component adaptation in which the regular pricing collaboration is being defined for the product component (cf. Figure 3). The composite adapter Reg_Pricing_Product in Figure 8 implements a nested class adapter for each role in the collaboration, namely LineItemRole, CustomerRole, ItemRole, ChargerRole and PricerRole. Each nested class adapter implements a collaboration role in terms of a Product component class, specified using the *adapts* relation. In this example, we assume the abstract pricing framework of Figure 2 has not yet been developed. Thus, we are intentionally hardwiring the regular pricing algorithm within the Reg_Pricing_Product adapter - the methods price() within LineItemRole and additionalCharge(...) within ItemRole.

Note how the adaptation body within nested class adapters, e.g., LineItemRole has three implicit self references: LineItemRole.this, Quote.this, and Reg_Pricing_Product.this. For instance, the price() method in LineItemRole calls operations defined in LineItemRole itself, e.g., pricer(), while item() calls the product() operation defined in Quote. If there were composite adapter level method or variable definitions they could as well be referred to from within any adaptation body via Reg_Pricing_Product.this. As long as there are no ambiguities self references remain implicit. For instance, we simply call product() within the implementation of item() in LineItemRole, implicitly meaning Quote.this.product(). If there are ambiguities, as in the case of the quantity() operation which is defined in both LineItemRole and Quote, we override the default precedence of the self reference, i.e., LineItemRole.this by explicitly delegating to the Quote.this self reference.

In contrast to the other two self variables, Reg_Pricing_Product.this has a second role in addition to serving as a name binding environment. The second role of Reg_Pricing_Product.this is in the implicit type lifting/lowering within the adapter scope. For illustration, consider the implementation of item() in the LineItemRole class adapter. One might expect a type mismatch between the result of invoking product() - assuming it to be HWProduct - and the return type of item(), which is ItemRole. However, there is no such conflict. This is because within Reg_Pricing_Product the type ItemRole is defined as an adaptation of the base type HWProduct. As the result, the composite adapter views any HWProduct that comes into its scope, e.g., the return value of calling product(), as automatically acquiring its item role in the pricing collaboration - the base object returned by product() will be automatically lifted to the expected ItemRole.

The key point in understanding type lifting/lowering is that individual class adapters make sense only within the scope of the enclosing adapter. The latter is not merely a syntactic construct for encapsulating adaptations for the classes involved in a collaboration (in general, a business process). The enclosing class adapter has an important semantic function: it defines a

"functor" that converts types from the base component's domain to the types defined within the nested class adapters (and thus to the types in the super component's domain in the case of component gluing) and vice versa. In other words, the composite adapter *lifts* objects from the base component's domain as they come into its scope, and subsequently *lowers* them as they leave its scope. In the following, we informally describe the semantics of *type lifting / lowering* and the role that *A.this* plays in this process.

In addition to binding composite adapter level variable and method definitions, the composite adapter self reference *A.this* also provides an environment for resolving type references within the composite adapter. This type environment is defined by the adaptationsOf_A function in Figure 9 which maps types from the base component C_1's domain to sets of types in the composite adapter A's domain. Given, C_1.B, adaptationsOf_A(B) includes any A.R defined as an adaptation of either C_1.B itself or one of its superclasses in C_1. For instance, given the composite adapter Reg_Pricing_Product in Figure 8 (RPP for short), $\text{adaptationsOf}_{RPP}$(HWProduct) is the set { ItemRole, PricerRole }. Given the type environment *A.this* of a composite adapter A, type conversions within A obey the rules given in the following paragraphs for *type lifting* and *type lowering*.

Type lifting. Let b be an object of the base type C1.B and assume that b comes into the scope of an adaptation body nested within adapter A in a context where an object of type A.R (or of the supertype of A.R in C_2, if A.R is defined by an adapts ... extends construct) is expected. An object of base type C_1.B comes into the scope of an adaptation body as the result of either (a) invoking an operation that returns an object of type C_1.B, or (b) directly instantiating C_1.B. If A.R $\in$ $\text{adaptationsOf}_A(C_1.B)$ then b will automatically be lifted to the role type A.R. Otherwise, a compile-time error occurs.

For illustration, consider the methods item() and pricer() in LineItemRole in Figure 8. They both call product() operation from Quote. Both invocations of product() cause a HWProduct object to come into the scope of the composite adapter Reg Pricing Product. However, within item() the HWProduct object will be lifted to ItemRole since it comes into scope in a context where an ItemRole instance is expected, while within pricer() the same HWProduct object will be lifted to PricerRole since it comes into scope in a context where a PricerRole object is expected. Both liftings are possible, hence the code is valid, since both PricerRole and ItemRole are adaptations of HWProduct.

Lifting b: C_1.B to A.R means: (a) finding the most specific subtype of R in A, A.R', such that A.R' $\in$ $\text{adaptationsOf}_A(C_1.B)$ and (b) binding all three "*self variables*" implicitly referred to within the adaptation body of R, *R'.this*, its corresponding base "self" *B.this*, and the enclosing composite adapter instance within which the lifting takes place *A.this*. The relationship between *R'.this* and *A.this* is essentially the conceptual relation between a Java inner

class instance and the enclosing toplevel class instance. The question remains what is the semantics of the *B.this-R.this* binding. This binding is realized as an aggregation in the current implementation of the adapter construct. That is, an instance of R' is created and some adaptee reference of this instance is intialized with B.this. Other alternatives are discussed in the following section.

```
import product ;
adapter Reg_Pricing_Product {
    adapter LineItemRole adapts Quote {
        protected ItemRole item ( ) { return product ( ) ; }
        protected CustomerRole customer ( ){ return customer ( ); }
        protected PricerRole pricer ( ) {
            PricerRole pr = product ( ) ;
            pr . setQty ( Quote . this . quantity ( ) ) ;
            pr . setCustomer ( Quote . this . customer ( ) ) ;
            return pr ;
        }
        protected int quantity () { return Quote. this . quantity ();}
        public double price ( ) {
            double basicPrice = pricer ( ) . basicPrice ( ) ;
            double discount = pricer ( ) . discount ( ) ;
            double unitPrice = basicPrice - ( discount * basicPrice );
                return u n i t P r i c e +
                    item ( ) . additionalCharge ( unitPrice , quantity ( ) ) ;
        }
    }
    adapter CustomerRole adapts Customer { }
    adapter ItemRole adapts HWProduct {
        protected ChargerRole charge ( ) { return tax ( ) ; }
        double additionalCharge ( double unitPrice, int qty ) {
            return charge ( ) . cost ( qty , unitPrice, this ) ;
        }
    }
    adapter ChargerRole adapts Tax {
        protected double
            cost ( int qty , double unitPrice, ItemRole item ) {
            return taxCharge ( qty , unitPrice, item ) ;
        }
    }
```

```
    adapter PricerRole adapts HWProduct {
        private int qty ;
        private Customer customer ;
        public void setQty ( int quantity) { qty = quantity;}
        public void setCustomer ( Customer cust ) { customer = cust ;}
        protected double basicPrice ( ) { return regPrice ( ) ; }
        protected double discount ( ) {
            return regDiscount ( qty , customer ) ; }
    }
}
```

Figure 8: Regular pricing - dynamic component adaptation

Lifting b: C_1.B to A.R means: (a) finding the most specific subtype of R in A, A.R', such that A.R' ∈ adaptationsOf$_A$(C_1.B) and (b) binding all three "*self variables*" implicitly referred to within the adaptation body of R, *R'.this*, its corresponding base "self" *B.this*, and the enclosing composite adapter instance within which the lifting takes place *A.this*. The relationship between *R'.this* and *A.this* is essentially the conceptual relation between a Java inner class instance and the enclosing toplevel class instance. The question remains what is the semantics of the *B.this-R.this* binding. This binding is realized as an aggregation in the current implementation of the adapter construct. That is, an instance of R' is created and some adaptee reference of this instance is intialized with B.this. Other alternatives are discussed in the following section.

Finally, it should be noted that the composite adapter A "remembers" the result of lifting a base object b: C_1.B to its role A.R. That is, if b goes in and out of the scope of A several times during the execution of operations defined in A, it will not be lifted to different A.R role adapter objects, i.e., the invocation history in the context of A does not get lost. This is crucial as the class adapter may add role-based state to the base object.

C_1_K: the domain of classes defined in C_1
$\leq C_1$_K: the subtype hierarchy in C_1
A_CA: the domain of class adaptations defined in A
adaptationsOf$_A$: C_1_K → P(A_CA): a function defined for all B ∈ C_1_K by
adaptationsOf$_A$(B) = {R ∈ A_CA | B adaptedTo$_A$ R}
adaptedTo$_A$ ⊆ C_1_K × A_CA: a binary relation, defined for all B ∈ C_1_K and R ∈ A_CA by

C_1.B adaptedTo$_A$ A.R ⇔
∃ SB ∈ C_1_K: B ≤ C_1_K SB ∧
adapter R adapts SB [extends C_2.S] adaptation_body ∈ A

Figure 9: The composite adapter type environment

Type lowering. Any class adapter object r: A.R that was created as the result of lifting base object b:C_1.B to A.R should be lowered back to type C_1.B before it can leave the adapter scope. This occurs when within the adaptation body of some nested class adaptation the role object r is either (a) passed as a parameter into an operation defined in any of the classes in base component C_1, or (b) an operation defined in class C_1.B is directly invoked on the role object r. Lowering r back to type C_1.B results in the original base object b.

For illustration, consider the implementation of cost(int qty, double unitPrice, ItemRole item) within ChargerRole in Figure 8. Passing cost's argument item as the third actual parameter to Tax.taxCharge(int qty, double unitPrice, HWProduct p) implies an automatic lowering of item: ItemRole to its base product.HWProduct object.

The commutative diagram in Figure 10, summarizes the relation between the type lifting and lowering performed within the scope of a composite adapter. In the diagram ASR (Adapter-level Structural Relation) stands for a structural relation between two types defined in a composite adapter, A.R and A.R', whereby the relation is realised via a structure mapping method. For instance, item: LineItemRole → ItemRole, item(li) = li.item(), is an example for an ASR (cf. Adapter in Figure 8). On the other hand, BSR (Base-level Structural Relation) is the (eventually computed) structural relation between two base types, C_1.B and C_1.B', where R ∈ adaptationsOf$_A$(B) and R' ∈ adaptationsOf$_A$(B'), that implements ASR. For instance, product: Quote → HWProduct, product(q) = q.product() is the base structural relation that implements the adapter structural relation item. Given b: C_1.B, the diagram imposes that BSR(b) = b' : C1.B' = lower(ASR(lift(b))).

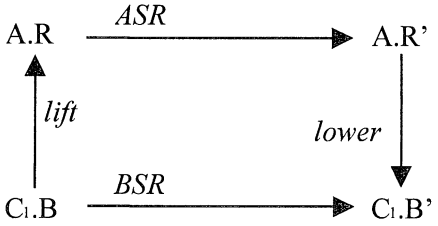


Figure 10: The relation between type lifting and lowering operations

Plugging adapters in at runtime. Given a composite adapter A that adds a collaboration to a component C1 and assuming that the root of the collaboration is a class adapter RR (for root role), a composite adapter instance a: A is dynamically applied to a base object b: C1.B, where RR ∈ adaptationsOf$_A$(C_1.B), by means of the liftTo operator (b liftTo a). Note that we are assuming that there is a single adaptation class for C_1.B in A. If there are several class adaptations for C_1.B in A, the application operator would

explicitly specify which one to use: b liftTo a.aRoleTypeName. As the result of the adapter application, a new class adapter instance for RR gets created.

Assuming that the root role class adapter defines the public method m, we can invoke m on the base instance by executing the invocation expression (b liftTo a).m(). The adapter a: A transforms the base objects of component C_1 that are encountered during the execution of method m into their lifted types in a "lazy way": new class adapter instances are created only when base objects come into a's scope. All class adapter instances created this way share the same composite adapter.

```
public class Client{
    static public void main ( String [ ] args ) {
        Quote q = new Quote (...) ;
        Reg_Pricing_Product rpp = new Reg_Pricing_Product ( ) ;
        System . out . println ( ( q liftTo rpp ) . price ( ) ) ;
    }
}
```

Figure 11: Class adaptation

For illustration, Figure 11 contains sample client code that dynamically plugs in the functionality of the Reg_Pricing_Product adapter. This allows the regular pricing functionality to be available to the quote object q.

3.2 Dynamic Component Gluing

Given components C_1 and C_2, the existence of an *extends* clause in a nested class adaptation { R adapts C_1.B extends C_2.S ... } puts the classes C_1.B and C_2.S in a target-adaptee relation in terms of the adapter design pattern [5]. The nested class adaptation R expresses the relationship C1.B plays-the-role-of C_2.S. The adaptation body defined in class adapter R encodes the delta by which we would have to enhance the definition of C_1.B if we used static subclassing to express the C_1.B plays-the-role-of C_2.S relationship.

For illustration, the composite adapter for gluing the concrete product application of Figure 3 and the abstract pricing framework component of Figure 2 according to the regular pricing scheme is given in Figure 12. While the composite adapter in Figure 8 directly implemented the regular pricing collaboration, the adapter in Figure 12 only customizes the abstract part of the pricing framework of Figure 2 in terms of the Product component and inherits the collaboration encoded by the methods price() and additionalCharge(...). Note again the implicit rebinding of types within the adapter scope. For instance, consider the return type ItemRole of item() and the return type HWProduct of the product() method that is called within item().

There is no conflict here because the type HWProduct of the object returned by product() will automatically be lifted to its extension HWProduct_ItemRole defined within the enclosing adapter, Reg_Pricing_Product, Since HWProduct_ItemRole is a subtype of ItemRole, the return type of item() is substitutable for the return type of product().

Note that in Figure 12 the class adapter PricerRole is nested within class adapter LineItemRole. Nesting class adapters into each other helps manage scoping issues. An inner class adapter has visibility for the definitions in the adaptation body of the enclosing class adapter, just like inner classes in Java can access definitions in the outer class. Clearly, nested class adapters can also be used for dynamic component adaptation - we could have made PricerRole in Figure 8 also a nested class adapter of LineItemRole. However, we preferred not to do so for the sake of keeping the discussion at that point as simple as possible and the reader's attention focused on the key features of the composite adapter construct, rather than being distracted by scoping issues.

In Figure 8 instance variables are used in the definition of PricerRole to refer to the customer and the quantity values, for which a PricerRole calculates the basic and discount prices. These variables are initialized when a new PricerRole is created by a LineItemRole. The same values are brought more elegantly into the scope PricerRole instances in the implementation in Figure 12. By making PricerRole an inner role of LineItemRole, each PricerRole instance automatically shares the item, customer, and quantity values of its enclosing LineItemRole instance. Both implementations are valid, since the design of the pricing framework in Figure 2 simply indicates that given the item, customer and quantity values of a line item, the PricerRole will calculate a basic price and a discount, while leaving open how these values are brought into the scope of the PricerRole.

3.3 The Adapter Construct at Work

In the following, we show how the adapter construct improves the modularity and extensibility of systems that result from assembling several components (business processes) in terms of our running example. Along the way, new features of the *pluggable composite adapter* model will be introduced as needed.

The main advantage of the adapter construct is that it facilitates the separation of concerns (functional ingredients of a system are separated from each other as well as the gluing concerns). This leads to improved modularity, hence, readability, maintainability, and more flexible extensibility. For illustrating these claims let us compare the effort required

to integrate a new pricing scheme (in general a new business process or multi-object collaborative functionality) when using traditional composition techniques versus an adapter construct. Assume that we have integrated the product and pricing components (cf. The class models in Figure 2 and 3) according to the regular pricing scheme. During this integration we have not anticipated integrations of alternative schemes. Thus, the class model will include only the original product classes modified as subclasses of the corresponding framework classes, along with the single adapter class HWProduct_PricerRole (cf. Figure 4).

```
import product ;
import pricing;

adapter Reg_Pricing_Product {
    adapter Quote_LineItemRole adapts Quote extends LineItemRole {
        protected ItemRole item ( ) { return product ( ) ; }
        protected CustomerRole customer ( ) { return customer ( ) ; }
        protected PricerRole pricer ( ) { return product ( ) ; }
        protected int quantity ( ) { return Quote . this . quantity ( ) ; }

    adapter HWProduct_PricerRole     adapts HWProduct
                                     extends PricerRole {
        public double basicPrice ( ) { return regPrice ( ) ; }
        public double discount ( ) {
            return regDiscount ( quantity ( ) , customer ( ) ) ;
        }
    }
}

    adapter Customer_CustomerRole adapts Customer
                                  extends CustomerRole { }

    adapter HWProduct_ItemRole adapts HWProduct extends ItemRole {
        protected ChargerRole charge ( ) { return tax ( ) ; }
    }
    adapter Tax_ChargerRole adapts Tax extends ChargerRole {
        public double
            cost ( int qty , double unitPrice , ItemRole item ) {
                return taxCharge ( qty , unitPrice , item ) ;
        }
    }
}
```

Figure 12: Regular pricing - dynamic component gluing

Now, assume we want to extend the system to support sale pricing. This extension will affect several places in the design in Figure 4, requiring a

subclass of HWProduct_PricerRole to be added, Quote to be modified to use a strategy object, a strategy class for PricingScheme to be added, along with the appropriate subclasses. Alternatively, in our model we can simply define a new composite adapter Sale_Pricing_Product by incrementally refining the existing composite adapter Reg_Pricing_Product that was given in Figure 12. The new adapter is shown in Figure 13.

```
import product ;
import pricing ;

adapter Sale_Pricing_Product extends Reg_Pricing_Product {

    adapter Quote_LineItemRole extends
        Reg_Pricing_Product . Quote_LineItemRole {

        adapter HWProduct_PricerRole adapts HWProduct
                                        extends PricerRole {
            public double basicPrice ( ) { return salePrice ( ) ; }
            public double discount ( ) { return 0;}
            }
        }
}
```

Figure 13: Sale pricing composite adapter

Adapters as extensions of other adapters. The meaning of the extends relationship between adapters is similar to the extends relation between classes in Java. Defining an adapter SA as a subadapter of adapter A means that SA (a) inherits all nested class adapters from A, (b) can define new nested class adapters, and (c) can override and/or incrementally refine class adapters nested within A. For instance, Sale_Pricing_Product inherits the Customer_CustomerRole, Tax_ChargerRole and HWProduct_ItemRole class adapters defined in the adapter Reg_Pricing_Product, while refining the definition of Quote_LineItemRole and overriding the HWProduct_PricerRole class adapter nested within Quote_LineItemRole.

One can factor out the commonalities of all pricing schemes in an abstract adapter - Pricing_Product in Figure 14. Reg_Pricing_Product is then defined as an extension of it, by (a) incrementally refining the Quote_LineItemRole adapter with an implementation for pricer() and (b) defining a new class adapter, HWProduct_PricerRole nested within Quote_LineItemRole. A composite adapter is abstract if (a) one of its nested class adapters is abstract, or (b) there is an abstract class in the super collaboration for which no class adapter is defined within the composite adapter. For instance, Pricing_Product in Figure 14 is abstract because the

class adapter Quote_LineItemRole is abstract (the abstract method pricer() from LineItemRole remains unimplemented).

At the beginning of this section, we mentioned that it is preferable that a component construct be a closed entity with well-defined interfaces. One of the benefits of this feature is that the adapter compiler can check whether an adapter is abstract based on the criteria (a) and (b) above.

Adapting an adapter. Adapters can also adapt other adapters, a feature that becomes very handy when business processes are layered on top of each other. An illustrative scenario is the following. Assume that after we have extended the bare business model of our hardware products supplier (cf. Figure 3) with the pricing functionality (cf. Figure 2), the integrated system needs to be further extended with the ability to calculate the total price of a given Order. Furthermore, assume that the generic component (a mini-framework) whose class model is given in Figure 15 is available for calculating the sum of a certain value in a composite object structure. Summing is modeled in the diagram in Figure 15 as the template method sum() in Composite, while what is summed is left open (the method value() in Elements is left abstract), as it will vary in different contexts where the summing component might get used.

We would like to reuse the summing component in calculating the total of (regular, sale, or negotiated) price of an order. Finally, an order is a composite of quotes and after having integrated the pricing framework into the product package, we know that quotes can be lifted into line items, hence we can calculate their price which will be the value to sum. To integrate the summing component into the system (consisting of the product component integrated with the pricing collaboration) we define the new composite adapter Summing_Pricing_Product given in Figure 16.

Note the declaration of Summing_Pricing_Product as an adaptation of the previously defined abstract composite adapter Pricing_Product. A composite adapter A that adapts another composite adapter BA dynamically extends BA instances with new nested class adapters, which may adapt either (a) BA's base component's classes, or (b) BA's class adapters themselves. For instance, Summing_Pricing_Product defines two class adapters: Order_Composite adapts the Order class in the base component of Pricing_Product, while Quote_Element adapts the class adapter Quote_LineItemRole defined in Pricing_Product.

```
import product ;
import pricing ;

abstract adapter Pricing_Product {
    abstract adapter Quote_LineItemRole adapts Quote
                                        extends LineItemRole {
        protected ItemRole item ( ) { return product ( ) ; }
        protected CustomerRole customer ( ) { return customer ( ) ; }
        protected int quantity ( ) { return Quote . this . quantity ( ) ; }
    }
    adapter Customer_CustomerRole adapts Customer
                                    extends CustomerRole {
        / as in Figure 12 /
    }
    adapter HWProduct_ItemRole adapts HWProduct
                                  extends ItemRole {
        / as in Figure 12 /
    }
    adapter_Tax ChargerRole adapts Tax extends ChargerRole {
        / as in Figure 12 /
    }
}
adapter Reg_Pricing_Product extends Pricing_Product {
    adapter Quote_LineItemRole
            extends Pricing_Product . Quote_LineItemRole {
        protected PricerRole pricer ( ) { return product ( ) ; }

        adapter HWProduct_PricerRole adapts HWProduct
                                        extends PricerRole {
            / as in Figure 12 /
        }
    }
}
```

Figure 14: Extension of abstract pricing composite adapter

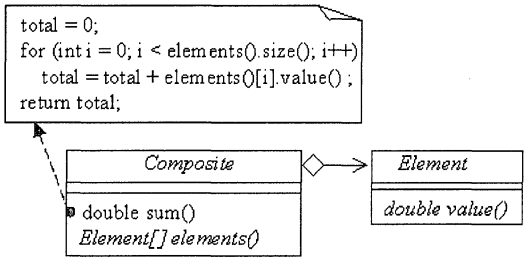


Figure 15: Summing component

```
import product ;
import summing ;

adapter Summing_Pricing_Product adapts Pricing_Product {
    adapter Order_Composite adapts Order extends Composite {
        Elements [ ] elements ( ) { return quotes ( ) ; }
    }
    adapter Quote_Element adapts Quote_LineItemRole
                                extends Element {
    double value ( ) { return price ( ) ; }
    }
}
```

Figure 16: Integration of summing with pricing and product

Note that a top-level *adapts* relation has similar dynamic semantics as the class-level adapts relation. That is, instances of concrete subadapters of Pricing_Product will appear to dynamically acquire class adaptations defined in Summing_Pricing_Product. Any composite adapter instance of type Pricing_Product that gets lifted to the type Summing_Pricing_Product is enabled to (a) view any instance of the base type Order within its scope as being of type Order_Composite, i.e., of type Composite with a particular implementation of the abstract method elements() (a feature that is not supported by a "pure" Summing_Pricing_Product composite adapter), and (b) view any class adapter instance of type Quote_LineItemRole within its scope as automatically acquiring the ability to also play the role of a summing Element as defined by Quote_Element.

This results, e.g., in a double type lifting performed on the elements of the array returned by the invocation of quotes() within the implementation of elements() in Order_Composite. As defined in the class product.Order, the method quotes() returns an array of Quote objects. However, since quotes() is being executed within the scope of a composite adapter of type Pricing_Product each Quote object q contained in the returned array gets automatically lifted to a ql: Quote_LineItemRole. The latter (ql: Quote_LineItemRole) gets further lifted to a qe: Quote_Element, since the composite adapter within whose scope it is created is not a pure Pricing_Product, but rather one that is lifted to the type Summing_Pricing_Product.

We could have as well used the extends rather than the adapts relationship between adapters in order to define Summing_Pricing_Product. However, adapts is in this case preferable in order to avoid extending all Pricing_Product subadapters. Using the static extends rather than the dynamic adapts relation would result in (a) a proliferation of adapters: Summing_Reg_Pricing_Product, Summing_Neg_Pricing_Product, Summing_Sale_

Pricing_Product, and (b) duplication of the code in the adaptation body of Summing_Pricing_Product in Figure 16.

Using the adapts relation rather than extends allows us to reuse the adaptation encoded within the body of Summing_Pricing_Product with instances of all subadapters of Pricing_Product. This is illustrated by the sample client code in Figure 17.

```
public class Client {
    static public void main ( String [ ] args ) {
        Order o = new Order ( ) ;
        Reg_Pricing_Product regpp = new Reg_Pricing_Product ( ) ;
        Sale_Pricing_Product salepp = new Sale_Pricing_Product ( ) ;
        Summing_Pricing_Product summpp = new Summing_Pricing_Product ( ) ;
        ...
        System . out . println ( ( o liftTo ( regpp liftTo sumpp ) ) . total ( ) ) ;
        System . out . println ( ( o liftTo ( salepp liftTo sumpp ) ) . total ( ) ) ;
    }
}
```

Figure 17: Using the Summing_Pricing_Product adapter

Recall that (b liftTo a) causes the base object b to be lifted to the role defined for its class in the composite adapter A. Thus, in Figure 17 we first "adapt" the Reg_Pricing_Product adapter regpp by the Summing_Pricing_Product adapter sumpp. The resulting adapter is then used to lift the Order object, with the total() method invoked on the adapted order object. A similar process applies for sales pricing.

In the discussion so far the base component has been a concrete application. Adapters can also be used to glue together two abstract collaborations, as illustrated in Fig 18. Now, we first glue together the abstract collaborations defined in the summing and pricing packages, resulting in the abstract composite collaboration Total_Pricing and then glue the latter with the product package by specifying Total_Pricing_Product and its subadapters.

4. ALTERNATIVE REALIZATIONS OF THE ADAPTER CONSTRUCT

Alternative realizations of the adapter construct can be classified in two main groups:

Global scope. The modifications specified by the adapter are globally visible in the sense that after compiling adapter A { ... adapter R adapts C_1.B extends C_2.S adaptation_body . . . } only an in-place modified B is visible with the new definition being equivalent to class B extends S { B-def adaptation

body}. If only single inheritance is supported and B already uses the sole inheritance link, one should think of the resulting definition of B as being equivalent to class B {B_defs S_defs adaptation_body}, i.e., S's definition after flattening its inheritance chain gets "copied" within B. This is the generative case that applies the adapter statically, i.e., at adapter compilation time or during class loading. Either source code generators or binary component adaptation tools [6, 3] can be used for this purpose. The client's use of the integrated functionality is as in the sample code for the pricing example in Figure 19. Note the regular- prefix of the price() method. This is because we assume a code generation scheme that uses adapter names to qualify method names for avoiding name clashes.

```
import pricing ;
import summing ;
import product ;
abstract adapter Total_Pricing {
    abstract adapter Pricing_Composite extends Composite { }
    abstract adapter LineItemRole_Element adapts LineItemRole
                                                extends Element {
        double value ( ) { return price ( ) ; }
    }
}
abstract adapter Total_Pricing_Product extends Total_Pricing {
    adapter Order_Pricing_Composite adapts Order
                                          extends Pricing_Composite {
        Elements [ ] elements ( ) { return quotes ( ) ; }
    }
    adapter Quote_LineItemRole_Element adapts Quote
                                            extends LineItemRole_Element {
        / as in Pricing Product in Figure 14 /
    }
    adapter HWProduct_ItemRole adapts HWProduct extends ItemRole {
        / as in Pricing Product in Figure 14 /
    }...
}
adapter Reg_Total_Pricing_Product extends Total_Pricing_Product {
    / as in Reg Pricing Product in Fig 14 /
}
```

Figure 18: Integration of summing with pricing and product

The primary advantage of static, invasive integration is efficiency, as it will avoid the need for excessive delegation to the base object. Actually, there are no adapter-adaptee pairs and the functor role of the composite adapter instances is no longer needed; composite adapters do not exist during run-time. The primary disadvantage is loss of flexibility. Enforcing the component adaptation to occur at either compile or load time restricts a

runtime object from acquiring the services of new components that become available after its instantiation. Thus only anticipated collaborations are supported. This also contrasts the Java philosophy of loading program constructs as they are needed, and not before. Since the base component's classes are modified in-place, all class instances are affected. Thus, it is not possible to be selective as to which objects acquire the new class semantics. Furthermore, objects become very heavy, in that they must support the state required for all of the possible collaborations in which they might participate.

```
public class Client {
    static public void main ( String [ ] args ) {
        Quote q = new Quote ( ) ;
        System . out . println ( q . regular_price ( ) ) ;
    }
}
```

Figure 19: Using a globally scoped regular pricing

Local scope. The adapter introduces a new name space, which means that after the composite adapter is compiled both the original B as in the component C_1 and its role in the collaboration defined by the composite adapter, A.R will co-exist. Two subcases can be distinguished here:

1. *Transparent scope*: Given adapter R adapts B extends S adaptation_body defined within the composite adapter A, we have that A.R is substitutable for B. This would be e.g., the case if A.R is generated as class A.R extends S { B-defs adaptation-code }. That means, if b is an instance of B and a an instance of A, than (b liftTo a) provides the original interface of B plus the interface of A.R.
2. *Opaque scope*: A.R is a "view" over B, meaning that for object b we can call methods in the original interface of B, while the result of (b liftTo a) provides only the interface of A.R.

The composite adapter pattern presented in [16] (the structure of which was shown in Section 2) provides (Java) programmers with a language idiom to simulate the b.2 version of the adapter construct. A simulation of adapters with transparent scope (b.1) requires extended object models supporting object-based inheritance [11] as e.g., *Self* [24] or the *Darwin* model and its Java based realization called *Lava* presented in [7]. Assuming *Lava* as the underlying language, modifying the technique in [16] to support transparent scope essentially means replacing the *adaptee* relation between role and base objects in Figure 6, i.e., forwarding semantics, with a *delegatee* relation as defined in *Darwin*, i.e., true delegation semantics with late-binding of self. The rest of the pattern remains the same.

Recall though that the pattern presented in [16] is too low-level for the average programmer to manage. For experimental purposes, we have

developed a more abstract version of the technique presented in [16] that makes use of the reflective facilities of the Java 2 platform. Briefly, the technique is as follows. The programmer defines all composite adapter classes as direct or indirect subclasses of a predefined class CompositeAdapter. On the first call of its constructor a CompositeAdapter makes use of the reflective API of Java 2 -namely of the new method getDeclaredClasses() implemented in java.lang.Class - in order to automatically establish the infrastructure of adapter factories (cf. Figure 6) as well the infrastructure needed for managing the adapter-adaptee relations. In this way the work of the programmer is facilitated, in that he/she (a) does not need to implement the AdapterObject interface from Figure 6 for each inner class adapter, and (b) does not need to take care of defining and initializing the factory objects for each class adapter. However, he/she still needs to explicitly call lifting/lowering operations at the appropriate places. Further details about this realization are out of the scope of this paper. The interested reader can find more information and sample code for the examples in the paper in [14].

The prototype outlined above allows us to experiment with the adapter within the context of a mainstream language such as Java. But it still remains complex for an inexperienced programmer. In the long run, a preprocessor will be needed to allow the programmer to work with the high-level adapter construct presented in this paper, while having the preprocessor statically transform adapter bodies to generate the corresponding composite adapter class as it would be hand-written by the programmer, if he/she used the pattern described in [16].

5. RELATED WORK

VanHilst and Notkin propose an approach for modeling collaborations based on templates and mixins as an alternative to using frameworks [20]. However, this approach may result in complex parameterizations and scalability problems. Smaragdakis and Batory solve this by elevating the concept of a mixin to multiple class granularity, using C++ parameterized nested classes [17]. However, their approach does not address the issue of dynamic customizations as described by Holland [9]. A *Contract* [9] allows multiple, potentially conflicting component customizations to exist in a single application. However, contracts do not allow conflicting customizations to be simultaneously active. Thus, it is not possible to allow different instances of a class to follow different collaboration schemes.

Seiter et al. proposed a *context relation* to link the static and dynamic aspects of a class [18]. While supporting multiple dynamic collaboration

schemes, the approach is based on dynamically altering a class definition for the duration of a method invocation, thus affecting all class instances. Multiple dynamic variations of an object's behavior are also supported in the Rondo model [13]. However, Rondo does not provide explicit support for collaborations. In this paper we propose a model for scoping the different collaboration schemes, thus we can be selective as to which objects are affected.

Batory proposed the *GenVoca* architecture to define parameterized, plug-compatible, interchangeable and interoperable components [1]. The *GenVoca* model is based on the notion of *realm*, *interface*, *component* and *layer*. Layers represent encapsulations of composite-object decorators, which could be dynamically composed. The technique we have presented can be used as an elegant Java implementation for *GenVoca* layers. Einarson and Hedin also suggest the use of inner classes as alternative implementations of several design patterns [4].

Mattson et al. [12] also indicate the problems with framework composition, analyze reasons for these problems, and investigate the state of the art of available solutions. Bosch argues that language support should be provided for explicitly describing design patterns in object-oriented programs [2]. Among supporting other patterns, he also provides a language construct for specifying a class as the adapter of another class, i.e., for explicit expression of the adapter pattern [5]. The adapter construct as proposed in [2] has two main restrictions. First, it does not support adaptation of entire collaborative functionality. Second, as indicated in [2], it does not allow interface incompatibility.

An underlying theme of the work described in this paper is separation of concerns to avoid software tangling. This is also the motivation behind both *Aspect-Oriented Programming* [25] and *Hyperspaces* (a new model of subject-oriented programming) [19]. AspectJ [25] is an extension of Java that allows one to program different aspects separately. Mezini and Lieberherr proposed *Adaptive Plug and Play Components*, or AP&PCs, which define a slice of behavior for a set of classes, and can be parameterized to allow reuse with different class models. An enhanced form of AP&PCs that decreases tangling of connectors and aspects in AspectJ is described in [10]. This improved form of AP&PC uses similar techniques as described in this paper, along with tool support.

Summary and Future Work. This chapter studied traditional framework customization techniques and concluded that they are inappropriate for component-based programming since they lack support for non-invasive, encapsulated, dynamic customization. We proposed a new language construct, called *pluggable composite adapter*, for expressing component gluing explicitly and discussed alternative realizations of the

construct. The construct allows the separation of customization code from application and framework implementations, resulting in better modularity, hence, in more flexible extensibility, easier maintenance and understandability.

As described in this paper, the adapter construct is focused only on a special kind of integration: *additive integration*. The adapter-based integration allows dynamic, noninvasive extension of a base component with additional behavior. In the more general case, the integration has overriding rather than additive semantics, i.e., it will cause the modification of existing behavior in at least one of the components being integrated. Weaving aspects as they are conceived in the aspect-oriented extension of Java, Aspect/J [25], has an overriding nature.

Although overriding semantics are not supported by the version of the adapters presented in this paper, there is nothing inherent that would prevent an appropriate extension of the adapter construct to serve this need. In fact, as preliminary work in that direction shows [10], adapters seem to provide an appropriate mean for decreasing tangling between aspect definitions and weaving concerns present in Aspect/J, providing for more reusable aspects.

This research track is one of our very near goals in the future. We are also working on a denotational semantics of the pluggable composite adapter construct. An investigation of how adapters play together with fundamental component technology such as the EJB model, Java's servlets, etc. will be an interesting area of future work. Last but not least, we intend to demonstrate the usefulness of the pluggable composite adapter construct by benchmarking it using real-life applications.

ACKNOWLEDGEMENTS

We would like to thank Sonali Kochar, David Lorenz, Doug Orleans, and Johan Ovlinger for their feedback on this paper. The second author was partially sponsored by the National Science Foundation under grant number CDA-972057, and the third author by the Defense Advanced Projects Agency (DARPA) and the Rome Laboratory under agreement number F30602-96-2-0239.

6. REFERENCES

1. Don Batory, Vivek Singhal, Jeff Thomas, Sankar Dasari, Bart Geraci and Marty Sirkin. The GenVoca Model of Software-System Generators. In IEEE Software, 11(5), 1994.
2. J. Bosch. Design Patterns as Language Constructs. In Journal of OOP, 1998.
3. G. Cohen, J. Chase, and D. Kaminsky. Automatic Program Transformation with JOIE. In USENIX 1998 Annual Technical Conference, pp. 167-178, 1998.

4. D. Einarson and G. Hedin. Using Inner Classes in Design Patterns. Available at <http://www.dna.lth.se/home/daniel/patterns inner classes.html>.
5. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.
6. Holzle and R. Keller. Binary Component Adaptation. In Proceedings of ECOOP '98, Springer Verlag LNCS 1445, pp. 307-329, 1998.
7. G. Kniesel. Type-Safe Delegation for Run-Time Component Adaptation. In Proceedings of ECOOP '99, Springer Verlag LNCS 1628, pp. 351-366, 1999.
8. J. Gil and D. Lorenz. Design Patterns and Language Design. In IEEE Computer, 31(3), pp. 118-120, 1998.
9. I. Holland. The Design and Representation of Object-Oriented Components. Ph.D. Dissertation, Northeastern University, Computer Science, 1993.
10. K. Lieberherr, D. Lorenz, and M. Mezini. Modeling Aspects with Adaptive Plug & Play Components. College of Computer Science, Northeastern University, Technical Report No. NU-CCS-99-01, Boston, MA, 1999.
11. H. Lieberman. Using Prototypical Objects to Implement Shared Behavior in OO Systems. In Proc. of OOPSLA '86, ACM Sigplan Notices, 21(11), pp. 214-223, 1986.
12. M. Mattson, J. Bosch and M. Fayad. Framework Integration Problems, Causes, Solutions. Communications of ACM, 42(10), pp. 80-87, 1999.
13. M. Mezini. Variational Object-Oriented Programming Beyond Classes and Inheritance. Kluwer Academic Publishers, 1998.
14. M. Mezini. A Reflective Implementation of Pluggable Composite Adapters. http://www.informatik.uni-siegen.de/ mira/DynCompGlue.html
15. M. Mezini and K. Lieberherr. Adaptive Plug and Play Components for Evolutionary Software Development. In Proceedings of OOPSLA '98, 33(10), pp. 97-116, 1998.
16. L. Seiter, M. Mezini, K. Lieberherr. Dynamic Component Gluing in Java. In Proc. of 1st Symposium on Generative and Component-Based Software Engineering (GCSE '99), Springer Verlag, LNCS, 1999.
17. Y. Smaragdakis and D. Batory. Implementing Layered Designs with Mixin Layers. In Proceedings of ECOOP'98, Springer Verlag LNCS, pp. 550-570, 1998.
18.] L. Seiter, J. Palsberg, and K. Lieberherr. Evolution of Object Behavior using Context Relations. In IEEE Transactions on Software Engineering, 24(1), pp. 79-92, 1998.
19. P. Tarr, H. Ossher, W. Harrison, S. Sutton Jr. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In Proceedings of ICSE'99, pp. 107-119, 1999.
20. M. VanHilst and D. Notkin. Using Role Components to Implement Collaboration-Based Designs. In Proceedings of OOPLSA'96, pp. 359-369, 1996, San Jose.
21. IBM San Francisco. http://www.software.ibm.com/ad/sanfrancisco/about.html
22. R. Monson-Haefel. Enterprise Java Beans. O'Reilly & Associates, Inc., 1999.
23. C. Szyperski. Component Software: Beyond Object-Oriented Programming. Addison-Wesley, 1998.
24. D. Ungar and R. Smith. Self: The Power of Simplicity. In Proceedings of OOPSLA '87, ACM Sigplan Notices, 22(12), pp. 227-242, 1987.
25. Xerox PARC AspectJ Team. AspectJ, Xerox PARC Technical Report, January 1999. http://www.parc.xerox.com/spl/projects/aop/

Chapter 12

ASPECT COMPOSITION USING COMPOSITION FILTERS

Lodewijk Bergmans, Mehmet Akşit and Bedir Tekinerdoğan
TRESE group, Department of Computer Science, University of Twente, P.O. Box 217, 7500 AE, Enschede, The Netherlands. email: {bergmans, aksit, bedir}@cs.utwente.nl, www: http://trese.cs.utwente.nl

Key words: composition, aspects, multiple views, view partitioning, view extension, view refinement, history sensitiveness, synchronization, composition filters

Abstract: This chapter first discusses a number of software reuse and extension problems in current object-oriented languages. For this purpose, a *change case* for a simplified mail system is presented. Each evolution step in the change case consists of the addition or refinement of certain aspects to existing classes. These examples illustrate that both inheritance and aggregation mechanisms cannot adequately express certain aspects of evolving software. This deficiency manifests itself in the number of superfluous (method) definitions that are required to realize the change case. As a solution to these problems, the composition filters model is introduced. We evaluate the effectiveness of various language mechanisms in coping with evolving software as in the presented change case.

1. INTRODUCTION

One of the most important principles in software engineering is the separation of concerns principle [7]. This principle states that a given problem involves different kinds of concerns, which should be identified and separated to cope with complexity and to achieve the required engineering quality factors such as adaptability, maintainability, extendibility and reusability. Despite a common agreement on the necessity of the application of the separation of concerns principle, its application for large and complex

applications that involve multiple concerns may become problematic. The reason for this is that concerns that are separated at the design level may become scattered at the programming code level and be tangled with implementations of other concerns within methods [10].

Problems in composing various concerns have been described in different publications. For example, in [12, 13, 4] the conflicts between the implementation of synchronization concerns and inheritance in object-oriented concurrent programming languages are described. In [3] the problems in composing concerns for real-time specifications are discussed. In [1, 6] the so-called *multiple views* composition problems have been addressed. In all these cases, a *conceptually sound* composition cannot be adequately expressed in a given language. The term *inheritance anomaly* was coined in [12, 13] to denote a more specific case of *composition anomaly* where the embedding of synchronization code in method implementations causes unnecessary redefinitions if the synchronization code has to be reused and/or extended through inheritance. In those cases, it typically appears that the problems can be patched by overriding in a subclass substantial parts of the methods defined by its superclass. This conflicts, however, with the intended reuse and causes reduced maintainability.

In section 2 we will illustrate various composition anomalies by using an example change scenario of a simple mail system. Using this example, we show that the conventional object-oriented composition techniques cannot deal with certain concern compositions satisfactorily. In section 3 we will introduce the composition filters model, which is an extension to the object-oriented model. Composition filters offer a better support for reusing and extending software with certain concerns, for example that are presented in the mail system case. In section 4 we will provide the composition filters solution to the composition anomalies that have been addressed in section 2. Finally, we will provide our conclusions in section 5.

2. EXAMPLE: DESIGN OF A MAIL SYSTEM

Figure 1 shows the class diagram of a simple mail system, which consists of classes *Originator*, *Email*, *MailDelivery* and *Receiver*. Class *Email* represents the electronic messages sent in this system and provides methods for defining, delivering and reading mails. For example, methods to write and read the attributes *originator*, *receiver*, and *content* of a mail object. The methods *putRoute()*, *getRoute()*, *deliver()* and *isDelivered()* are used by class *MailDelivery* while routing and delivering the messages from originators to receivers. The method *reply()* is used by receiver objects to send a reply

message. In this text, *Email* will be used as the base class that can be specialized into various kinds of email objects.

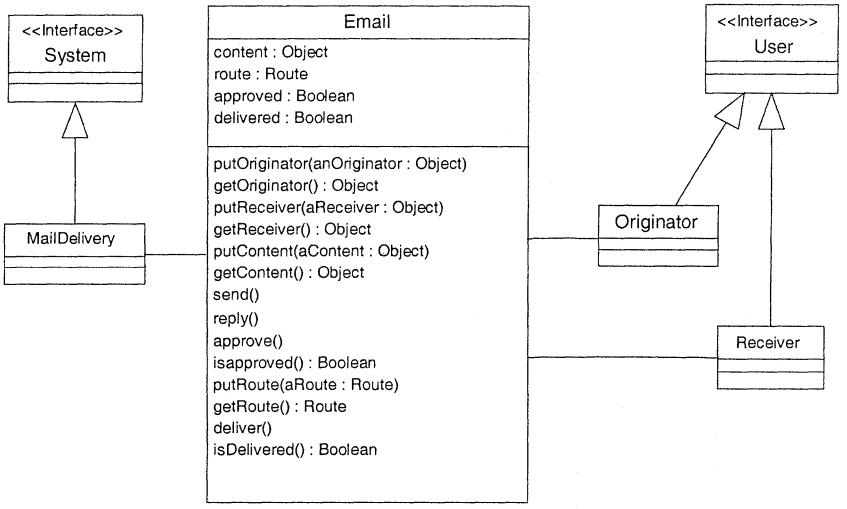


Figure 1: The interface methods of class *Email*

To illustrate a number of composition anomalies, class *Email* will be extended to support additional concerns. The concerns that we will address are the following:

1. *adding multiple views*, whereby the access to the mail interface is distinguished for a user and system view.
2. *view partitioning*, whereby the existing views are partitioned into additional sub-views.
3. *view extension*, whereby the views are extended.
4. *history sensitive behavior*, whereby information on history is logged.
5. *synchronization to multiple classes*, whereby locking mechanisms are added to multiple classes.

The motivation for these change cases is to show that it is in many cases impossible to define such extensions in the object-oriented model without superfluous redefinitions, i.e. composition anomalies.

Object-orientation provides two different mechanisms for composing concerns; either through aggregation or through inheritance (see also chapter 2, section 2.2.5). For each change case, we will discuss the application of both mechanisms.

2.1 Adding Multiple Views

The current implementation of class *Email* allows any client object to access e.g. the contents of a mail. We specialize class *Email* into *USViewMail* (User/System-View) and restrict access to its methods based on the class of the client object (i.e. the object that was the sender of the invocation). If the client is of the *User* type (i.e. an *Originator* or a *Receiver*), it is allowed to execute the methods *putOriginator()*, *putReceiver()*, *putContents()*, *getContents()*, *send()* and *reply()*. The methods *approve()*, *putRoute()* and *deliver()* are used by the clients of the *System* type (i.e. an instance of class *MailDelivery*). No restrictions are required for the other methods.

We assume that the identity of the client object (the sender of the message) can be obtained[1]. The following two subsections discuss aggregation-based and inheritance-based approaches within the conventional object-oriented model as supported by programming languages such as Java, C++ and Smalltalk.

In the case of aggregation-based composition, the *USViewMail* object encapsulates an instance of class *Email* and implements the view checking operations *userView()* and *systemView()*[2]. For each method that requires a view constraint to be enforced, additional code must be inserted that implements this constraint. Because the methods have already been implemented in class *Email*, invoking the appropriate method in the encapsulated *Email* object reuses this implementation. For example method *putOriginator()*, which is subject to the 'User' view can be implemented as follows in pseudocode:

```
USViewMail::putOriginator(Object anOriginator)
   if self.userView()   // returns true if view applies
   then return imp.putOriginator(anOriginator)
   else self.viewError();
```

The class diagram in Figure 2 shows aggregation-based composition of multiple views. Notice that in this implementation strategy, all the methods have to be declared and implemented by class *USViewMail*, even those methods that do not require any view enforcement. This is because, these methods must be accessible through the *USViewMail* object.

[1] Note that in most language implementations this is far from trivial, if not impossible. For example in Smalltalk and Java there are –computationally expensive– ways to access the identity of the client through the calling stack.

[2] We implement view checking in separate methods for the purpose of reuse.

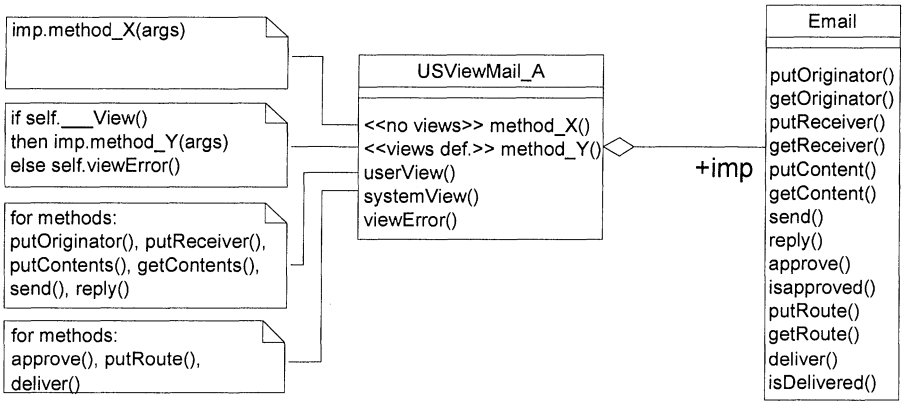


Figure 2: Aggregation-based composition of multiple views

Figure 3 provides the class diagram for the inheritance-based composition. View checking is again implemented at the start of each view-constrained method, reuse is now realized through *super* calls. Only the methods that require views have to be redefined; other methods can be inherited from the superclasses.

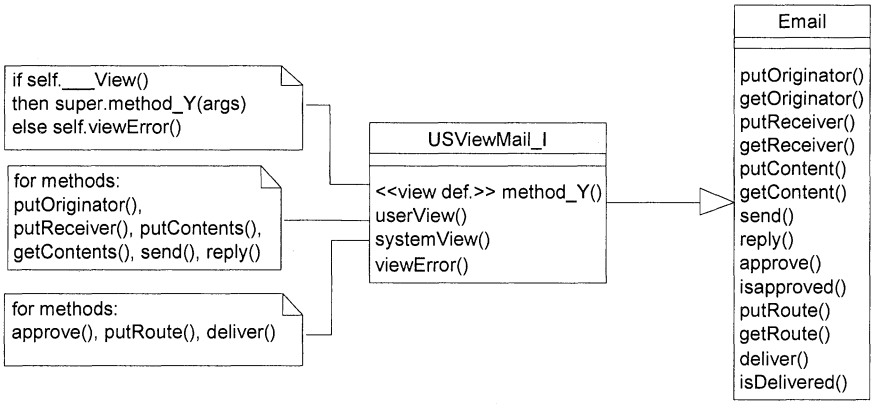


Figure 3: Inheritance-based composition of multiple views

The method *putOriginator()* is implemented as follows:

```
USViewMail::putOriginator(Object anOriginator)
  if self.userView()
  then return super.putOriginator(anOriginator)
  else self.viewError();
```

Adopting aggregation-based composition, *USViewMail* implements 16 methods. Among these, 9 methods implement view checking and forwarding, 5 methods are used for forwarding only, and 2 methods implement the views (excluding the *viewError()* method). The inheritance-based implementation requires 11 methods. Here, 9 methods implement view checking and super class calls and 2 methods implement the views. Ideally, we should only implement the two view implementation methods and a mapping between these methods and the methods to which they apply (i.e. can be prefixed). The following table summarizes these numbers:

Table 1: Evaluation of composition anomalies in *USViewMail*

Composition Scheme	# Method (re-)definitions
Ideal/Conceptual	2+ 1 view mapping
Aggregation	16
Inheritance	11

2.2 View Partitioning

Assume that class *ORViewMail* partitions the *User* view into *Originator* and *Receiver* views. Only *originator* clients are allowed to invoke the methods *putOriginator()*, *putReceiver()*, *putContent()* and *send()*. *Receiver* clients are only allowed to invoke the method *reply()*. For other methods, the restrictions (if any) defined by *USViewMail* apply.

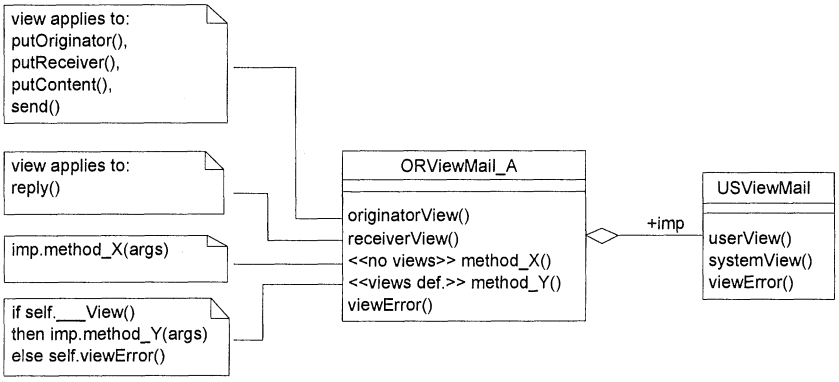


Figure 4: Aggregation-based composition of view partitioning

Again, this specification can be implemented using aggregation or inheritance-based composition. In the example, in case of aggregation-based composition, the aggregated object is an instance of class *USViewMail*. The implementation is along the same lines as for class *USViewMail*, as shown in Figure 4. This means that all the methods for which the additional

constraints defined by *originatorView()* and *receiverView()* apply, must be redefined to add this new view constraint. All other methods that must appear on the interface of class *ORViewMail* have to be defined and redeirected as well.

In the inheritance-based composition approach, class *ORViewMail* inherits from class *USViewMail*. This requires redefining all the methods that are subject to the newly defined *originatorView()* and *receiverView()*. All other methods are inherited from class *USViewMail*. This implementation is shown in the following figure:

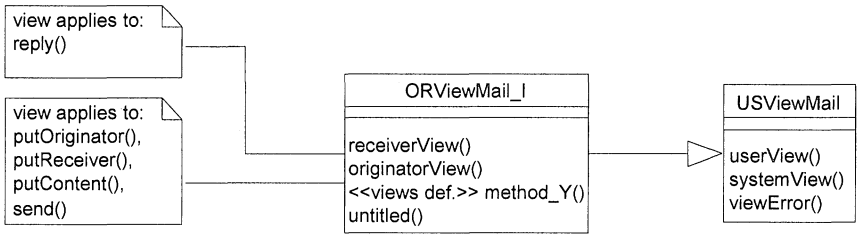


Figure 5: Inheritance-based composition of view partitioning

We now summarize the number of method (re-)definitions required for the view partitioning. Ideally, we only have to define the two views, and specify the methods upon which these apply (rather than embedding the view checking inside each method implementation). In the aggregation-based case, for each reused method that must be visible on the interface (i.e. 14 methods), a redirecting method must be created. For all of these methods upon which the views apply, also the view checking must be embedded in these method implementations. In addition, two new methods defining the *originator* and *receiver*-view must be created. In the inheritance-based case, the two new views must be implemented, and all the methods that require one of these views (respectively 1 and 4) must be redefined as well. The following table shows these numbers:

Table 2: Evaluation of composition anomalies in *ORViewMail*

Composition Scheme	# Method (re-)definitions
Ideal/Conceptual	2+1 view mapping
Aggregation	16 (+2 view redirection methods)
Inheritance	7

Note that in the aggregation-based approach, the view definitions of *userView* and *systemView* are not directly available for class *ORViewMail*.

2.3 View Extension

In the next example, we extend the *originatorView()* and *receiverView()* that are defined in class *ORViewMail* so that they apply to a *group of originators* and *receivers*. This may be required, for example, in offices where more than one person is responsible for sending and receiving mails.

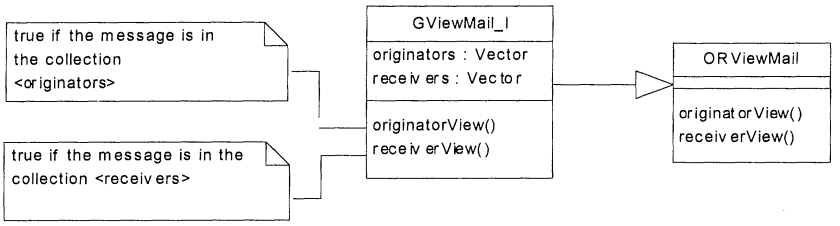


Figure 6: Inheritance-based composition of view extension

In inheritance-based composition, the methods *originatorView()* and *receiverView()* of class *ORViewMail* are overridden in *GViewMail* to define *group originator* and *receiver* views, respectively. All other methods can be inherited from class *ORViewMail*. A call to, for example, *originatorView()* in method *ORViewMail::send()*, will then refer to the *originatorView()* implemented in *GViewMail*. In this way, only 2 methods are required for re-implementing the views (we assume that *UserView()* and *SystemView()* need not be re-implemented). *Figure* shows inheritance-based composition.

In aggregation-based composition, the implementation of class *GViewMail* is analogous to that of *ORViewMail* as shown in Figure 6. The two methods that implement the views are redefined, the methods that are subject to a view must include the checks to these new view methods[3], and the other reused methods require a simple redirection implementation.

Table 3: Evaluation of composition anomalies in GviewMail

Composition Scheme	# Method (re-)definitions
Ideal/Conceptual	2
Aggregation	16
Inheritance	2

The inheritance-based approach behaves in this case ideal: only the new view conditions require additional method definitions. In the aggregation-based approach, in total 16 methods have to be implemented: 5 methods are

[3] This is because in the aggregation-based case we do not have the equivalence of dynamic binding through a *self* pseudo-variable.

used for view checking, 9 methods are used for forwarding messages only, and 2 methods implement the views. Table 3 shows these numbers.

2.4 History Sensitive Behavior

Assume that we want to introduce the following refinement: a class *HistoryMail*, which adds a view that does not depend on the client object, but upon historical information about the invocations on class *HistoryMail*. If the same method is invoked twice or more in a row for the same mail object, a warning (error) message must be generated, and the method is not executed. Assume that this constraint applies to the methods *send()*, *reply()*, and *deliver()*.

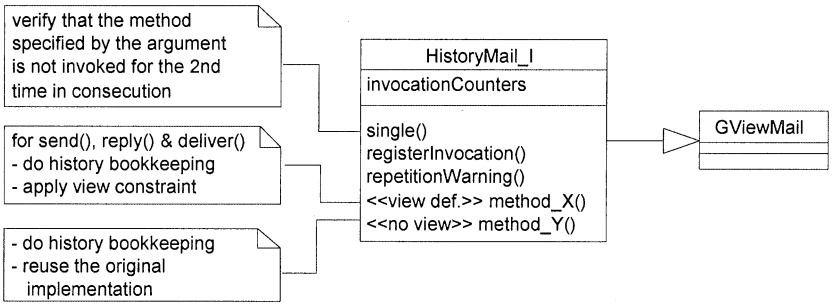


Figure 7: Inheritance-based composition of history sensitive behavior.

Figure 7 shows inheritance-based composition of the new history-sensitive behavior with class *GViewMail*. The realization of this behavior involves two conceptually different problems: the first is to collect the relevant history information. This requires bookkeeping of all the method invocations, including those that have no constraint defined upon them. In other words, bookkeeping of invocations is an aspect that applies to all the methods of an object and affects seemingly unrelated parts of the class. The bookkeeping requires redefinition of all the reused methods, for example as illustrated by the following pseudo-code for the inheritance-based strategy:

```
HistoryMail::methodY(<args>)  // only add bookkeeping
   self.registerInvocation('methodY');
   return super.methodY(<args>);
```

The second issue is to enforce the constraint upon the three selected methods. This requires –a different– redefinition of all the (three) methods

that are subject to the history-sensitive behavior, for instance as shown in the following pseudo-code example:

```
HistoryMail::methodX(<args>)
  // history-sensitive; add bookkeeping snd verify view
  self.registerInvocation('methodX');
  if self.single('methodX')
  then return super.methodX(<args>)
  else self.repetitionWarning();
```

The aggregation-based solution is almost identical; all the method implementations are as shown above, except that in each place where "*super.methodX()*" is written in the inheritance-based case, the aggregation-based case will have "*imp.methodX()*" instead. Both cases require 3 new methods (*single()*, *registerInvocation()* and *repetitionWarning()*) and a total of 14 methods to be redefined (that excludes all view and bookkeeping methods defined by the reused classes).

Extension with history-sensitive behavior introduces two new aspects to the class:

- *History bookkeeping aspect*: this aspect *crosscuts* all the methods of the class: it requires the addition of (a call to) bookkeeping code to all the methods. Thus one can image a composition scheme (and language model) that requires only the definition of the bookkeeping (as in the method *registerInvocation()*) and a specification that states that this definition applies to all methods of the class, including the inherited methods and –most likely– the methods that will be introduced in subclasses.
- *Constraint* behavior *aspect*: for the three methods *send()*, *reply()*, and *deliver()*, the constraints have to be specified (i.e. they are not executed twice in a row, and then an error is generated). Ideally we would like to specify this constraint only once and then simply assign it to the three methods.

The following table shows the number of definitions the above solutions require. Since all the solutions require the history bookkeeping implemented by all methods, the number of methods that are redefined are the same in all solutions.

Table 4: Evaluation of composition anomalies in *HistoryMail*

Composition Scheme	# Method (re-)definitions
Ideal/Conceptual	2 + 2 mappings
Aggregation	3 new +14 redef.
Inheritance	3 new +14 redef.

2.5 Adding Synchronization to Multiple Classes

Our final example deals with the extension of the mail example with a code necessary to synchronize concurrent threads. It is different from the previous extensions in that we want to extend multiple classes: *HistoryMail*, *MailDelivery*, and *Receiver*. However, all of these classes need to be extended with the same logical feature: the ability to *lock* all operations such that they are halted until an *unlock* operation has been called. We introduce class *SyncMailSystem*, from which we want to reuse the synchronization specification and 2 additional operations called *lock* and *unlock*. If the method *lock* is invoked, then *all* subsequent messages are delayed until the invocation of the method *unlock*.

To illustrate a possible implementation we use *semaphores*, one of the simplest mechanisms to delay and activate threads[4]. A first question is how to extend the three classes: the approach taken so far in this chapter is to create specialized classes. In this case this means the introduction of three new classes, namely *SyncMail*, *SyncMailDelivery* and *SyncReceiver*. Because these three classes require similar extensions, we focus on one class, i.e. *SyncMail*: the same changes have to be repeated for the other two classes.

SyncMail must reuse behavior from both *HistoryMail* and *SyncMailSystem*. With inheritance, this requires support for multiple inheritance, 'multiple aggregation' is equivalent to repeated aggregation and requires no special language support. Because all the (reused) methods of *SyncMail* are affected, aggregation-based and inheritance-based composition are further largely identical: each method that is visible on the interface must be extended to start with some code that verifies the locking state and acts accordingly, for example:

```
SyncMail::methodX()        // any method of SyncMail
   if impSyncMS.isLocked() then impSyncMS.wait();
   return impHM.methodX();
     // call original method from instance of HistoryMail
```

[4] For the sake of simplicity, we will ignore possible concurrency conflicts: this could also be integrated with the locking code, but we assume they are handled by separate mechanisms.

This implementation calls the method *isLocked()* upon an instance of class *SyncMailSystem* (i.e. *impSyncMS*) to request the locking state. If the state is 'locked', the thread will be blocked by calling *wait()*, otherwise the original method is executed by an instance of class *HistoryMail*.

The implementation of *SyncMail* as illustrated in the previous section requires in total 16 method definitions. Here, 14 methods are overridden to add synchronization constraints, 2 methods are required to forward the *lock()* and *unlock()* operations. In the case of multiple inheritance, forwarding methods is not needed, which reduces the number of methods to be defined with 2.

Table 5: Evaluation of composition anomalies when adding synchronization.

Composition Scheme	# Method (re-)definitions
Ideal/Conceptual	4 new +1 mapping
Aggregation (multiple)	4 new + 24 redef. + 6 forw. = 34
Inheritance (multiple)	4 new + 24 redef. = 28

In addition, this must be repeated for classes *SyncMailDelivery* and *SyncReceiver*, if these have each 5 methods on their interface, the total number of defined methods is 14+2 + (5+2) + (5+2) + 4 = 34. The final number 4 refers to the number of new methods defined by class *SyncMailSystem*. Table 5 shows the number of definitions that are required for the various solutions.

All the previous evolution steps have illustrated the need to apply certain behavior repeatedly within other methods of the same class. This repetition characteristic is also referred to as *crosscutting* [10]. One of the distinctions in this case is that, in addition, behavior needs to be repeated within other classes. In other words, the synchronization crosscuts methods in multiple classes.

2.6 Overall Evaluation of Change Cases

The following table provides an overview of the number of method definitions for each of the classes and each of the two reuse strategies (i.e. aggregation respectively inheritance). The first column gives (an estimation of) the number of definitions that are at least needed to realize the concern that is to be introduced by each of the classes; i.e. in the ideal case. If the same behavior is to be applied to multiple methods, this may be expressed by defining a simple mapping between the definition of that behavior and the methods; we separately add the number of such mappings.

Table 6: An overview of the Mail change case and resulting anomalies

Extension (class)	Ideal	Aggregation		Inheritance	
	#defs	#defs	#anom	#defs	#anom
Email	14	14	0	14	0
USViewMail	2+1	16	14-1	11	9-1
ORViewMail	2+1	18	16-1	7	5-1
GviewMail	2	16	14	2	0
HistoryMail	2+2	17	15-2	17	15-2
SyncMail	4+1	34	30-1	28	24-1
Totals	26+5	115	89-5	79	53-5
	31	115	84	79	48

The table shows the number of definitions ('#defs') and 'the number of anomalies' ('#anom') for each of the strategies. The number of anomalies is equal to the number of superfluous definitions, as calculated by subtracting the number of definitions required in the ideal case from the actual number of definitions required.

We can conclude that from the perspective of reusability, the conventional object-oriented model –at least in the given example case– performs unsatisfactorily. The examples show that reusing components through aggregation and inheritance mechanisms may not always be successful, if objects implement concerns like multiple views, history information and synchronization. An important characteristic of the presented problems is that they involve crosscutting behavior.

The aggregation-based change case requires 84 superfluous (re-) definitions. Inheritance-based reuse performs better ('only' 48 superfluous definitions), but cannot implement dynamically changing reuse relations.

Despite of all these composability problems, the object-oriented model has many useful features. For example, the change case we presented shows that each of the versions of the mail system can be adequately realized; it is the evolution between versions that cannot be dealt with satisfactorily. For this and other –more practical– reasons, we believe that to cope with the evolution problems, we should enhance *current* object-oriented languages, rather than replacing them.

3. THE COMPOSITION FILTERS MODEL

3.1 Basic Structure of Composition Filters Objects

The composition filters (CF) model is a modular extension to the 'conventional' object model as adopted e.g. by Java, C++ and Smalltalk. The behavior of an object can be substantially affected and enhanced through the manipulation of incoming and outgoing messages only. To do so, in the CF model, a layer called the *interface part* is introduced. The resulting model and its components are shown in Figure 8.
The most significant components in the CF model are the *input filters* and *output filters*. Each individual filter specifies a particular manipulation of messages. Various filter types are available for different types of manipulations. The filters together compose the behavior of the object, possibly in terms of other objects. These other objects can be either *internal objects* or *external objects*. Internal objects are encapsulated within the composition filter object whereas external objects remain outside the composition filters object, such as globals or shared objects. The behavior of the object is a composition of the behavior of its internal and external objects.

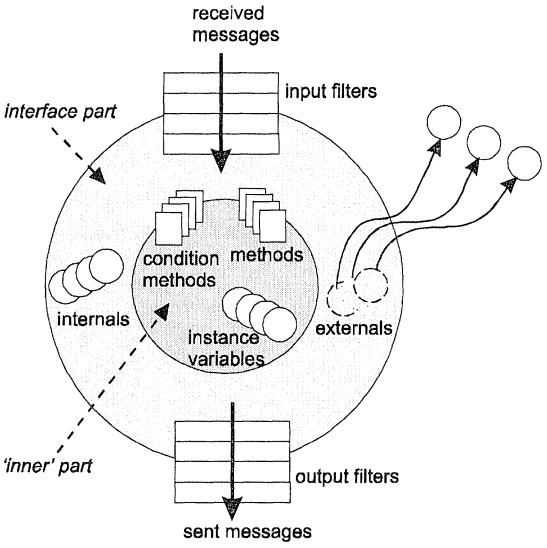


Figure 8: The components of the composition-filters model

In addition, –part of– the behavior of the object can be implemented by the 'inner' object, which is therefore also referred to as the *implementation part*. Any conventional object-oriented programming language, such as Java,

C++ or Smalltalk[5] can implement the inner object: the interface part is a modular extension to the inner object.

3.2 The Principle of Message Filtering

We will explain the basic mechanism of message filtering with the aid of Figure 9. The discussion focuses on input filters, but output filters work in exactly the same manner. The main difference is that output filters deal with messages sent by the object instead of received messages.

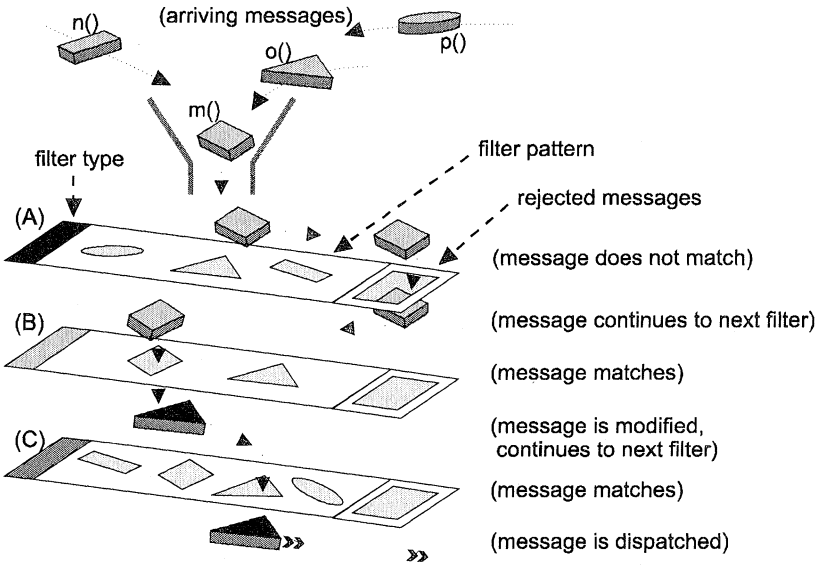


Figure 9: An intuitive schema of message filtering.

To understand the schema the following should be kept in mind: filters are defined in an ordered set. A message that is received by an object is first reified, i.e. a first-class representation of the message is created[6]. The reified message has to pass the filters in the set, until it is discarded or can be dispatched. Dispatching means that the message is activated again, for example to start the execution of a method body, or to be delegated to another object. Each filter can either accept or reject a message. The semantics associated with acceptance and rejection depend on the type of the filter.

[5] Implementations of composition filters have been built as extensions for each of these languages in the past [9, 11, 14].

[6] Composition filters thus apply a form of message reflection [8].

Figure 9 visualizes the processing of messages by three filters, A, B and C. An object can receive a variety of messages, in the figure exemplified by m(), n(), o() and p(). All received messages are subject to manipulation by all successive filters. Each filter tries to match messages based on a specific pattern. All filters for defining these patterns use a common syntax. The matching process can be defined in terms of message properties, but may also depend on the current state of the object.

We follow the message m() as it passes through the filters. In Figure 9, message m() does not match with the pattern defined by filter (A). Thus, this filter rejects the message. In this example, the rejected message is simply passed on to the next filter.

The message will then be evaluated by filter (B). The pattern that is defined by this filter matches with the message. This is referred to as *acceptance* of the message by the filter. This initiates a particular action that depends on the filter type: the message may be manipulated and modified, other side effects might take place as well. In the example of filter (B), the message is modified (designated in the figure by its changed shape and color), and then passed on to the next filter.

For the last filter in the example, filter (C), the pattern also matches the message. The acceptance of the message in this case causes the message to be dispatched, for example to a local method of the object. The message itself contains information that determines how it should be dispatched (i.e. the target object and the message selector).

In general, every filter set should contain a filter that causes messages to be dispatched, as this is the only means to trigger the execution of a method. For output filters, dispatching means that the message is submitted to the target object. Note that also in this case, upon the reception of this message by the target object, the message must first pass the input filters of the target object.

3.3 Filter Specifications

In this section, we briefly introduce the syntax and intuitive semantics of filter specifications. A single filter specification consists of the following elements:

```
<name>:<filter-type>={<filter-elem>,<filter-elem>, … };
```

This can be interpreted as the declaration of a filter instance of <filter-type> with the name <filter-name>, which is initialized with an expression that contains a number of filter elements that are separated by comma's[7].

Filter elements take the following form[8]:

```
<condition> => [<match-target>.<match-sel>]<target>.<sel>
```

In this element, the condition can be seen as a guard that enables the rest of the element. The default condition is the *True* condition, which always enables the element. On the right hand side, matching (between the square brackets) and substitution (the rightmost pair) with the target and/or the selector of a message takes place. The 'implication' operator '=>' has a counterpart, expressed as '~>', which means that if the condition is satisfied and the message does match on the right-hand side, the filter element will reject the message.

As a simple example, consider the following filter expression that defines extension of class *Mail*, by inheriting from *Mail* and adding a number of new methods (the first filter element is redundant, for illustrative purposes only):

```
inh:Dispatch={True=>[outer.m]inner.m, inner.*, superObj.*};
```

The first element is always enabled by the condition *True*, then continues to select only the messages with selector *m* that have been sent to the interface of this object (*'outer'*). If this matches, the target of the message is replaced with *inner* and the message selector with *m* (which is redundant since this was already the case). The second element has no explicit condition, in which case the default condition *True* is assumed. As a result it will match with any message that is defined by (i.e. is in the signature of) *inner*, and in that case substitute *inner* as the target of the message. The third element has again the default condition *True*, and will thus match with any message in the signature of –the class of– *superObj*, and in that case substitute *superObj* as the target of the message.

If the message matches any of the filter elements, the resulting (modified) message will be dispatched, i.e. the method defined by the *selector* of the message is to be executed upon the object defined by the *target* of the message. If the target is *inner*, this causes direct method execution. But if the target is another composition filters object, the message is delegated to that object, and will start by evaluating the filters of that object.

[7] Actually, the comma's represent just one particular composition operator (best described as a conditional OR). Other operators have been discussed previously and may be defined and implemented in the future.

[8] Actually, both the left- and the right-hand side of the '=>' can consist of a set of the respective elements.

3.4 The Superimposition Mechanism

In this section we briefly introduce the superimposition mechanism of the composition filters model. This is an extension to the model that we have presented so far: Figure 10 shows a refined version of Figure 8 with a few new elements. In the place of the set of input filters in Figure 8, now a box with a number of *instantiations* of filters is shown: these sets of filters are defined elsewhere (in the same or other objects). This is exemplified by the gray *filter definitions* in the figure.

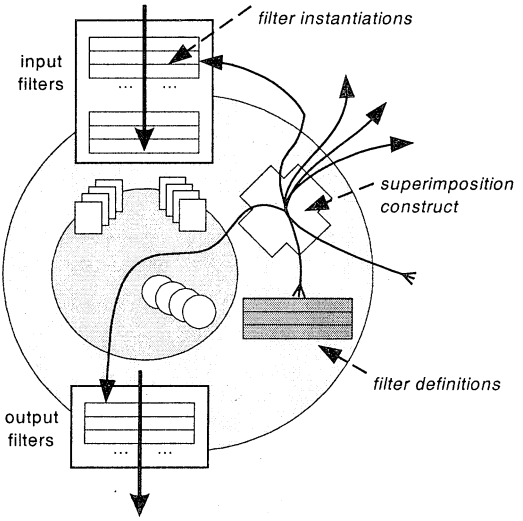


Figure 10: Superimposition mechanism in the composition filters model.

The most important part is the *superimposition* construct: this is a part of a class definition that maps the filtersets (from this object or others) onto a set of instances (possibly including instances of the current class). This means that the filterset is added (on top of) the existing filterset(s). The superimposition construct can also be applied to superimpose other elements of the interface part, such as internals, externals and conditions. For more details on the superimposition mechanism we refer to [5].

3.5 Summary of CF Principles

The composition-filters approach aims to enhance the expression power and maintainability of objects. Filters are based on the following principles:

1. There are a number of pre-defined filter classes, each responsible for expressing a certain aspect. Programmers may introduce new filters, provided these fulfil a number of requirements.
2. Instances of a filter class can be created and attached to a class defined in various languages such as Java [14], Smalltalk [11] and C++ [9]. This may occur and change dynamically (at run-time). However, dynamic changes may degrate understandability, correctness and implementation optimizations and therefore must be realized with care.
3. A filter instance is initialized using a filter expression. A standard filter expression syntax and semantics are available for all filters. This is a declarative specification, in the sense that it does not make any assumptions about how the specification is to be implemented[9].
4. A message manipulation operation by a filter may change the *explicit* and *implicit* attributes of the received message. The explicit attributes are the receiver object, the message selector and the arguments of the message. The implicit attributes include the sender and server object of the message, and other attributes that can be introduced by filters or application programmers.
5. A filter specification refers to the parameters of the received messages only. It does not make any assumption about other filters. However, a filter may refer to the state of its object, as made accessible and abstracted through the *conditions* of the object.
6. A filter expression consists of a set of filter elements. These elements and/or filters themselves can be composed using logical operators such as conditional-OR, conditional-AND, and exclusion. In the composition filters syntax, the character "," implements a conditional-OR operation, which means that if the expression on the left-hand-side cannot match, then the expression on the right-hand-side will be evaluated. A conditional-AND operation can be implemented by cascading filters, using the ";" sign in the filter definition language.
7. Each filter expression specifies a single concern, which is then mapped upon one or more messages that are executed by a method of some object (in particular the object itself). This implements the specification of crosscutting concerns, although with a scope that is restricted to the local object and the objects that is explicitly delegated to.
8. Superimposition of filters upon groups of objects can be used to express concerns that crosscut multiple classes. Superimposition does not break the encapsulation of objects, but only relies on public interfaces.

[9] A filter and its parts can be implemented in various ways, for example, as run-time objects by adopting message reflection (e.g. in [11]), or as in-lined code, by adopting compilation and optimisation techniques (e.g. in [14]).

9. Typing is based on signatures that are derived for each object from its filter specification. For type checking purposes, the filter interface definition language may require additional type declarations, e.g. of objects that are reused.

4. COMPOSITION FILTERS APPROACH TO THE MAIL PROBLEM

4.1 Multiple Views

The composition filters version of class *USViewMail* has two attached (input) filters. The filter *USView*, which is an instance of an *Error* filter, expresses multiple views. If an *Error* filter accepts the received message, then it is forwarded to the following filter. Otherwise an exception is generated. The filter *execute* is an instance of a *Dispatch* filter. If a *Dispatch* filter accepts the received message, then the message is executed. The interface (filter) definition of this class can be written as follows:

```
inputfilters
   USView : Error =
      { UserView    => {putOriginator, putReceiver,
                        putContent, getContent, send, reply},
        SystemView => {approve, putRoute, deliver},
         True        => {getOriginator, getReceiver,
                         isApproved, getRoute, isDelivered} };
  execute : Dispatch = { inner.*, mail.* };
```

The conditions *UserView* and *SystemView* are Boolean methods defined by class *USViewMail*. If *UserView* is true, then the *Error* filter accepts the messages *putOriginator*, *putReceiver*, *putContent*, *getContent*, *send* and *reply*. Similarly, the messages *approve*, *putRoute* and *deliver* are only accepted if *SystemView* returns true. The remaining 5 methods are not restricted by the *Error* filter, because the condition is specified as *true*.

The specifications "inner.*" and "mail.*" in the *Dispatch* filter mean that the filter accepts *all* (cf. wildcards) the methods declared by class *USViewMail* and the class of the internal mail object: *Email*. The pseudo-variable *inner* refers to the inner part of the current instance of *USViewMail*.

Since filters are separated and largely independent from the class, they can be reused separately. For example, software engineers can implement the core functionality of the classes mentioned above in any object-oriented language without attaching filters. Filters can later be stacked and attached to

any of these classes, whenever necessary. Note that the composition-filters implementation of *USViewMail* requires only 3 new definitions: 2 view implementations (the conditions) and 1 composition-filters specification (*USView*) to solve the view problem[10].

4.2 View Partitioning

The following filter definitions are required to realize class *ORViewMail* using composition filters:

```
ORView:Error =
       { origView => {putOriginator, putReceiver, putContent,
                      getContent, send},
         recView  => reply,
         true     ~> {putOriginator, putReceiver, putContent,
                      getContent, send, reply} }
execute: Dispatch = { inner.*, mail.* };
```

If the condition *origView* is true, the *Originator* view is valid and the messages *putOriginator*, *putReceiver*, *putContent*, *getContent* and *send* are accepted. These messages will then be dispatched to object *mail*, an instance of class *USViewMail*. If *USViewMail* is also extended with filters, the accepted message will pass through the filters of *USViewMail* object as well. The condition *recView* is used to enforce the *receiver* view; if this condition is true, the *reply* messages are accepted by the filter. The operator "~>" in the last part of this filter means that if the condition is true (which is always the case in this example), all messages are accepted except the specified ones. The effect of this is that all these other messages will always pass the filter, regardless of the actual view that applies. The composition-filters implementation of *ORViewMail* requires only 3 new definitions. These are the implementation of 2 views by conditions and the *ORView* filter specification.

4.3 View Extension

The composition-filters implementation of class *GViewMail* does not require any specific filter definition. Since conditions are methods, they can be reused from class *ORViewMail* or overridden if necessary. In *GViewMail*, these conditions can be redefined to check for groups of originators and receivers.

[10] We do not count the *Dispatch* filter because it is only used to express inheritance, something that we did not count as a separate definition in the examples of the conventional object model either.

4.4 History Sensitive Behavior

Consider the following definition of filters for class *HistoryMail* :

```
check : Meta = { [*]inner.verify }
bookkeeping : Meta = { [*]inner.count };
execute: Dispatch = { True=>{inner.*, mail.*} };
```

The *Meta* filter is used to reify a message. If the received message matches –in this specification it always matches because of the wildcard "[*]"–, it is reified. The resulting object is sent as the argument of a newly created message, with a target and selector as specified by the second part of the filter element.

In this case, the first *Meta* filter sends the reified message to the *inner* object, executing the method *verify*, which verifies repeated execution and generates a warning whenever appropriate. The second *Meta* filter sends the reified messages to the inner *bookkeeping* method, which performs the actual bookkeeping of the last executed message.

More detailed information about Meta-filters can be found in [2]. The composition-filters implementation of *HistoryMail* requires 4 new definitions: two filter specifications, and the methods *count* and *verify*.

4.5 Adding Synchronization to Multiple Classes

The problem of adding synchronization can be split in two issues: the first is how to specify synchronization, the second is how to attach this crosscutting specification to the three classes involved (*HistoryMail*, *MailDelivery*, and *Receiver*).

Using composition filters, we can express synchronization by a filter of type *Wait*; filters of this type perform synchronization of messages by queuing all messages as long as they cannot match with any of the filter elements. Locking can be expressed with the following filter definition:

```
queue: Wait = { True=>unlock, Unlocked =>* };
```

The message *unlock()* will always match at the first element, and is thus never blocked. If the condition *Unlocked* is true, then any message matches and will proceed to the next filter, otherwise all messages –except *unlock()*– will be queued until the condition *Unlocked* does become true.

Instead of creating three new subclasses, we can create a single class *SyncMail*, which contains all the new definitions and a superimposition specification to attach the necessary synchronization specifications to classes HistoryMail, MailDelivery and *Receiver* as well. The following specification shows how to superimposes the filterset *locking* (which contains the *queue*

filter of type *Wait* that we showed above) upon all the instances of classes *HistoryMail*, *MailDelivery* and *Receiver*:

```
superimposition
  selectors
  lockables={*=HistoryMail, *=MailDelivery, *=Receiver};
  filtersets
    lockables <- locking;
```

Superimposition specifications consists of two distinct parts: first one or more *selectors* are declared; each selector expression defines a set of instances. The second part uses the selector identifiers to superimpose filtersets (or internals, externals, conditions or methods) upon a certain set of instances as designated by the selectors.

This composition-filters implementation requires 6 new definitions. These are the filterset locking (with only a *Wait* filter specification), the condition *unlocked*, the two methods *lock()* and *unlock()*, the definition of the selector *lockables* and the superimposition of the filterset *locking*. In the case of an ideal or minimum number of definitions, the selector and superimposition could be merged[11].

4.6 Evaluation

In order to compare the composition filters approach with the conventional object model, we have counted the number of (method, filter or condition) specifications in each of the different change cases. They are shown in Table 7, together with the results from Table 6 that show the results for inheritance and aggregation.

Table 7: The Mail change cases and resulting anomalies including composition filters

Extension (class)	Ideal	Aggregation		Inheritance		Compos. filters	
	#defs	#defs	#anom	#defs	#anom	#defs	#anom
Email	14	14	0	14	0	14	0
USViewMail	3	16	13	11	8	3	0
ORViewMail	3	18	15	7	4	3	0
GviewMail	2	16	14	2	0	2	0
HistoryMail	4	17	13	17	13	4	0
SyncMail	5	34	29	28	23	6	1
Total definitions	30	115	**84**	79	**48**	32	**1**

This table is different from Table 6 only in the last two columns, where the results for composition filters are shown. It appears that the composition filters implementations are 'ideal' in the sense of the amount of redefinitions

[11] But note that this separation is made for reasons of modularity and adaptability.

that are required to implement these change cases. In short, this is due to the appropriate granularity and modularity that composition filters allow. The next section discusses the contributing factors in more detail.

5. CONCLUSION

In this chapter we have illustrated the limitations of the two most widely used forms of object composition in object-oriented design and programming: inheritance and aggregation. The change case of the mail system illustrates some of the problems in the evolution of object-oriented software. Each change case consists of the addition or refinement of a single aspect –or concern– to existing classes.

We have assumed the following requirements for dealing with evolving software:

- Modularity: the newly introduced aspect must be modeled as a separate entity of development and reuse.
- No modifications to the existing classes are allowed (this is partly implied by the previous item).
- Avoid code replication, because of the maintenance problems this brings (it also requires extra work).

The change case of the *Mail* example has introduced the following features (generalizations of these problems have been added between brackets):

- adding multiple views (i.e. dynamically adding constraints to groups of methods)
- view partitioning (or in general, state partitioning)
- view extension (i.e. condition refinement)
- history sensitive behavior (administering executions plus state dependent constraints)
- adding synchronization (adding special execution semantics to groups of methods in multiple classes)

Through these examples, we have illustrated that both inheritance and aggregation cannot *adequately* express certain cases of evolving software. This is apparent by looking at the number of definitions that were required: for inheritance 79, and for aggregation 115. In these cases 48 respectively 84 definitions were unnecessary from an 'ideal', i.e. conceptual point of view. These superfluous definitions are a serious maintenance problem.

In section 3 we briefly introduced the composition filters model: a modular extension to the object-oriented model that allows for composing new classes from (a) the visible behavior of existing classes and (b) well-

defined semantic actions. The latter semantic actions are defined by the various available *filter types*, and may for example express synchronization, exceptions, message reification and message dispatch.

We have shown in section 4 how the composition filters model can be used to support software evolution as illustrated by the Mail system change case. For the given example, almost no superfluous definitions were required to implement it. The following characteristics of the composition-filters model contribute most to this result.

- The composition mechanism of composition filters merges most of the benefits of both the aggregation-based approach and the inheritance-based approach.
- Composition filters provide abstractions (conditions) for expressing states (which can also be used to express views or constraints).
- Composition filters provide abstractions for mapping these states plus a certain behavior to one or more elements in the interface (signature) of an object. This realizes an important form of crosscutting of behavior over methods.
- Using the *Meta* filter, meta-level state such as history information can be obtained and managed in a straightforward way.
- The superimposition construct allows to specify a certain behavior (e.g. as a filter) in one place, and apply that behavior to multiple locations (classes). This expresses crosscutting across multiple classes.

One may wonder whether the example in this chapter suits the abilities of the composition filters model particularly well. Although there is an obvious match, the examples can be easily generalized (as described earlier in this section) to a very wide range of problems. The important benefits of the model lie in the composition mechanism, which is applicable to arbitrary domains.

ACKNOWLEDGEMENTS

This research has been supported and funded by various organizations including Siemens-Nixdorf Software Center, the Dutch Ministry of Economical affairs under the SENTER program, the Dutch Organization for Scientific Research (NWO, 'Inconsistency management in the requirements analysis phase' project), the AMIDST project, and by the IST Project 1999-14191 EASYCOMP.

6. REFERENCES

1. M. Akşit, L. Bergmans & S. Vural, *An Object-Oriented Language-Database Integration Model: The Composition-Filters Approach*, ECOOP '92 Conference Proceedings, LNCS 615, Springer-Verlag, 1992, pp. 372-395.
2. M. Akşit, K. Wakita, J. Bosch, L. Bergmans and A. Yonezawa, *Abstracting Object-Interactions Using Composition-Filters*, In: *Object-based distributed processing*, R. Guerraoui, O. Nierstrasz & M. Riveill (eds), LNCS, Springer-Verlag, 1993, pp 152-184.
3. M. Akşit, J. Bosch, W. van der Sterren, L. Bergmans, *Real-Time Specification Inheritance Anomalies and Real-Time Filters*, ECOOP'94 Conference Proceedings, LNCS 821, Springer-Verlag, 1994, pp. 386-407.
4. L. Bergmans. *Composing Concurrent Objects*, Ph.D. thesis, University of Twente, The Netherlands, 1994
5. L. Bergmans and M. M. Akşit, *Composing Crosscutting Concerns using Composition Filters*, Communications of the ACM, Vol. 44, No. 10, (to appear) October 2001
6. S. de Bruijn, *Composable Objects with Multiple Views and Layering*, MSc. thesis, Dept. of Computer Science, University of Twente, March 1998
7. E.W. Dijkstra. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, NJ, 1976.
8. J. Ferber, *Computational Reflection in Class-Based Object-Oriented Languages*, OOPSLA '89, ACM SIGPLAN Notices, Vol. 24, No. 10, October 1989, pp. 317-326
9. M. Glandrup, *Extending C++ using the concepts of Composition Filters*, MSc. thesis, Dept. of Computer Science, University of Twente, November 1995
10. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, J. Irwin, *Aspect-Oriented Programming*. In proceedings of ECOOP '97, Springer-Verlag LNCS 1241, June 1997.
11. P. Koopmans, *On the design and realization of the Sina compiler*, MSc. thesis, Dept. of Computer Science, University of Twente, August 1995
12. S. Matsuoka, K. Wakita & A. Yonezawa, *Synchronization Constraints with Inheritance: What is Not Possible- So What is?*, Tokyo University, Internal Report, 1990
13. S. Matsuoka & A. Yonezawa, *Inheritance Anomaly in Object-Oriented Concurrent Programming Languages*, in *Research Directions in Concurrent Object-Oriented Programming*, (eds.) Agha, Wegner & Yonezawa, MIT Press, April 1993, pp. 107-150
14. C. Wichman, *ComposeJ: The Development of a Preprocessor to Facilitate Composition Filters in the Java Language*, MSc. thesis, Dept. of Computer Science, University of Twente, December 1999.

INDEX