Lecture Notes in Computer Science 1951

Frank van der Linden (Ed.)

Software Architectures for Product Families

International Workshop IW-SAPF-3
Las Palmas de Gran Canaria, Spain, March 2000
Proceedings

Springer

Lecture Notes in Computer Science 1951
Edited by G. Goos, J. Hartmanis and J. van Leeuwen

Springer
Berlin
Heidelberg
New York
Barcelona
Hong Kong
London
Milan
Paris
Singapore
Tokyo

Frank van der Linden (Ed.)

Software Architectures for Product Families

International Workshop IW-SAPF-3
Las Palmas de Gran Canaria, Spain, March 15-17, 2000
Proceedings

Springer

Series Editors

Gerhard Goos, Karlsruhe University, Germany
Juris Hartmanis, Cornell University, NY, USA
Jan van Leeuwen, Utrecht University, The Netherlands

Volume Editor

Frank van der Linden
Philips Medical Systems
Veenpluis 4-6, 5684 PC Best, The Netherlands
E-mail: Frank.van.der.Linden@philips.com

Cataloging-in-Publication Data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Software architectures for product families : international workshop IW SAPF 3, Las Palmas de Gran Canaria, Spain, March 15 - 17, 2000 ; proceedings / Frank van der Linden (ed.). - Berlin ; Heidelberg ; New York ; Barcelona ; Hong Kong ; London ; Milan ; Paris ; Singapore ; Tokyo : Springer, 2000
(Lecture notes in computer science ; Vol. 1951)
ISBN 3-540-41480-0

CR Subject Classification (1998): D.2.11, D.2, K.6

ISSN 0302-9743
ISBN 3-540-41480-0 Springer-Verlag Berlin Heidelberg New York

Springer-Verlag Berlin Heidelberg New York
a member of BertelsmannSpringer Science+Business Media GmbH

Printed in Germany

Typesetting: Camera-ready by author, data conversion by PTP-Berlin, Stefan Sossna
Printed on acid-free paper SPIN: 10780953 06/3142 5 4 3 2 1 0

Preface

This book contains the proceedings of a third workshop on the theme of Software Architecture for Product Families. The first two workshops were organised by the ESPRIT project ARES, and were called "Development and Evolution of Software Architectures for Product Families". Proceedings of the first workshop, held in November 1996, were only published electronically at: "http://www.dit.upm.es/~ares/". Proceedings of the second workshop, held in February 1998, were published as Springer LNCS 1429.

The ARES project was finished in February 1999. Several partners continued cooperation in a larger consortium, ITEA project 99005, ESAPS. As such it is part of the European Eureka Σ! 2023 programme. The third workshop was organised as part of the ESAPS project. In order to make the theme of the workshop more generic we decided to rename it "International Workshop on Software Architectures for Product Families". As with the earlier two workshops we managed to bring together people working in the software architecture of product families and in software product-line engineering.

Submitted papers were grouped in five sessions. Moreover, we introduced two sessions, one on configuration management and one on evolution, because we felt that discussion was needed on these topics, but there were no submitted papers for these subjects. Finally, we introduced a surveys session, giving an overview of the present situation in Europe, focussed on ESAPS, and in the USA, focussed on the SEI Product Line Systems Program.

The workshop was chaired by Henk Obbink from Philips Research and by Paul Clements of the SEI.

The Programme Committee was recruited from the participants of the earlier two workshops and from partners within ARES and ESAPS:

Felix Bachmann	Philippe Kruchten
Bob Balzer	Jeff Maggee
Len Bass	Nenad Medvidovic
Jan Bosch	Robert Nord
T.W. Cook	Dewayne Perry
Juan Carlos Dueñas	Juan Antonio de la Puente
Wolfgang Emmerich	Alexander Ran
Loe Feijs	Clemens Szyperski
Martin Griss	Bernhard Thomé
Mehdi Jazayeri	Will Tracz
Jean Jourdan	David Weiss

Because of our good experiences with the second workshop, this workshop was held at the same place as the previous one: Las Palmas de Gran Canaria. The local organisation was again very well done, and in the hands of the same people as the last time: Juan Carlos Dueñas from the Universidad Polytécnica de Madrid and Javier Miranda from the Universidad de Las Palmas de Gran Canaria. Again it was a great place to have a workshop and the local organisation was done very well. The workshop itself was very satisfactory and addressed very well the needs of the participants to have a forum to discuss their experiences in software family development.

It is felt that the series of IW-SAPF workshops have to be continued. A next workshop is planned in the fall of 2001, at a different place, as most participants prefer to meet in different scenery.

October 2000

Frank van der Linden

Table of Contents

Product Family Methods

Evolution

Product Family Techniques

Surveys

Introduction

Frank van der Linden

Philips Medical systems B.V., Veenpluis 4-6,
5684 PC Eindhoven, the Netherlands
frank.van.der.linden@philips.com

These are the proceedings of the third workshop on the theme of Software Architecture for Product families. The series is originated by the drive of many companies to minimize both the costs of developing and the time to market of new members of a software rich product family (or product line). Today companies offering software-intensive systems have to face several trends: modularity, component technology, configurability, standardization, decreasing time to market, fast change of requirements, globalization, and more varying customer groups. These trends may cause serious problems if we don't meet them with adequate means.

This third workshop was organized as part of the Eureka Σ! 2023 programme, ITEA project 99005, ESAPS. The aim of this project is to provide an enhanced system-family approach in the following areas: Analysis, Definition and Evolution of system-families. ESAPS is designed to enable a major paradigm shift in the existing processes, methods, platforms and tools.

We believe that sharing maximization and reuse of software structure and components within the product family appear to be the path to follow. The primary focus of this workshop will be on methods, techniques and tools to manage the diversity of products in a family through an architecture point of view.

The aim of the workshop is to bring together professionals from academia and industry to exchange ideas, experiences and identify obstacles and propose solutions in the domain of software family architectures. Topics of interest include, but are not restricted to, the: business, organizational, product and process aspects for product families.

Submitted papers were grouped in 5 sessions. In addition, we introduced two sessions, on configuration management and on evolution, because we felt that discussion was needed on these topics, but there were no submitted papers on these subjects. Finally we introduced a surveys session, giving an overview of the present situation in Europe, mainly focussed upon ESAPS, and in the USA mainly focussed upon the SEI Product Line Systems Program.

A general impression is that we are getting mature. There is less confusion about terminology. During the workshop it appeared that the emphasis on process and organization issues. Technology is much less an issue. A poll under the attendants of the workshop showed an interest in continuation of the series. We are presently considering to organize it in the fall of 2001.

There were 49 participants:

- 41 from Europe
- 8 from the USA

F. van der Linden (Ed.): IW-SAPF-3, LNCS 1951, pp. 1-3, 2000.

- 29 from industry
- 20 from research institutions.

Sessions

The workshop was split into the following sessions:

1. Product Family Practice
   In this session a couple of experiences with the use of product family architectures were presented.
2. Business
   This session was meant to deal with the economic potential of product families.
3. Configuration Management
   This session was introduced because we felt a need to deal with configuration management from a product-line perspective. As it was already clear the SCM tools are not used in a uniform way and that no real support exists for product families.
4. Product Family Concepts
   This session was meant to deal with novel approaches towards product family development.
5. Product Family Methods
   This session was meant to deal with the global organization of product family development
6. Evolution
   This session was introduced because we felt a need to deal with the evolution of product family development.
7. Product Family Techniques
   This session deals with specific techniques that can be used within product family development.
8. Surveys
   In this session a European and an USA survey of product family/product-line development were presented

Most sessions consisted of a set of introductory presentations after which a lively discussion followed. The results of these discussions were reported in the session overviews within these proceedings.

Results of the Workshop

Although we had more time for presentations than the previous time, there was still a lot of room for discussions. Not all discussion lead to a clear conclusion, but several observations can be made.

At several places product family initiatives are set up. In many cases the industry itself finds a need for going in the product family business. Universities and research institutes follow. This is mainly caused by the fact that large-scale development programs are not feasible at such research institutes. However, research institutes are a crucial factor in standardizing and consolidating the approaches. In particular this has

both happened in Europe and in the USA. In the first case the origin comes from the ARES, PRAISE and ESAPS projects. In the second case the Product Line Systems Program has a central catalyzing role.

Although it seems to be obvious that a product family approach will be beneficial in the case that you have to deal with a large amount of variation, no clear models exist that determine the benefits of introduction of a product family programme, however, the first approaches are already available. A main reason for this may be that it the technology seems already to be available, but the organization and the development process have to be adapted, in uncertain directions. The adaptations often go into a direction that is not in line with the culture that has grown in the last 30 years within software development. In particular, initiating a development process and/or organizational adaptation will lead to all kinds of problems recognized within change-management. According to the discussions at the workshop we can conclude that these kinds of problems are much larger that the technical ones.

The aim of the workshop was to bring together professionals from academia and industry to exchange ideas, experiences and identify obstacles and propose solutions in the domain of software family architectures. In this respect the workshop was a success. The most prominent people in this field were present. Those that were not present have shown interest.

We have to continue in this direction. In Europe we follow the track started with ARES and ESAPS. In the USA the SEI is a central force in our community. We have to grow to each other, since there are still some differences between the two approaches, for instance already in terminology. The term “product family” is in use in Europe, the term “Product-line” originates from the USA. Both mean almost the same, but not exactly. As we may expect the future workshops on this topic will less pay attention to the technical matters, and more to the organizational and process issues. We are considering setting up a 4th workshop on this topic.

Component Frameworks for a Medical Imaging Product Family

Jan Gerben Wijnstra

Philips Research Laboratories
Prof. Holstlaan 4, 5656 AA Eindhoven, The Netherlands
JanGerben.Wijnstra@philips.com

Abstract. In this paper we describe our experience with component frameworks in a product family architecture in the medical imaging domain. The component frameworks are treated as an integral part of the architectural approach and proved to be an important means for modelling diversity in the functionality supported by the individual family members. This approach is based on the explicit modelling of the domain and the definition of a shared product family architecture. Two main types of component frameworks are described, along with the ways in which they are to be developed.

Keywords: Component Frameworks, Plug-ins, Diversity, Product Family Architecture, Domain Modelling, Interfaces

1. Introduction

Products in various embedded system markets are becoming more complex and more diverse, and they must support easy extension with new features. Also important factors are a short time-to-market and low development costs. Such requirements must be met with a product family that covers a large part of the market. The development of a product family and its individual members (i.e. single products) can be supported by a shared family architecture. Like any other architecture, such an architecture must cover a number of quality attributes like performance, security and testability; see chapter 4 of [2]. An important additional quality attribute of such an architecture is support of diversity (related to modifiability). This paper focuses on component frameworks as means for dealing with diversity.

The work whose results are presented here was carried out in the context of the development of a medical imaging product family. The main characteristics of this product family are:

- only a relatively small number of systems are delivered in the field, and almost every system is different due to high configurability and customisability
- the delivered systems must be supported for a long time (10 to 15 years), and updates of mechanical, hardware and software components in the field must be supported by field-service engineers
- new features must have a short time-to-market, and the fact that the product family deals with a relatively mature market implies that customers will have high expectations and will request specific features
- high demands are imposed on the systems' safety and reliability, because if a system does not operate according to specification, it may be potentially dangerous to patients and personnel

F. van der Linden (Ed.): IW-SAPF-3, LNCS 1951, pp. 4-18, 2000.

- there are different development groups at different sites, each of which is responsible for the development of a sub-range of the total product family range

These characteristics must be tackled by applying reuse among the family members. This is achieved by developing a common component-based platform for all the family members, using component frameworks as a means for realising diversity. This paper focuses on the approach taken with component frameworks. The description of this approach comprises the following parts:

- the context, consisting of domain modelling and family architecting
- how component frameworks can help in approaching the diversity at hand
- the kinds of component frameworks used
- further elaboration of one kind of component framework
- the steps to be taken to arrive at such a component framework

These issues will be described in sections 0 through 0. Concluding remarks are to be found in section 0.

2. Setting Up a Product Family

The approach for the development of the medical imaging product family is based on the principles presented in [6]. This approach identifies two main development areas: one in which a common family platform is developed, consisting of AFE (Application Family Engineering) and CSE (Component System Engineering), and one in which family members are developed using the family platform, which is called ASE (Application System Engineering). The family members are developed by multiple development groups, each of which is responsible for a sub-range of the total family range.

In this section two AFE activities are briefly described, viz. domain modelling and setting up the product family architecture.

Domain Modelling

When setting up a product family, one should focus not only on the functionality of the first product, but on the complete set of known products and products envisioned for the family. For the medical imaging product family, a requirements object model has been set up that describes the concepts of the domain that are relevant for all the products belonging to the medical imaging product family, including the diversity in the domain. It is described with the UML notation and comprises about 100 class diagrams, 700 classes, 1000 relationships, and 1500 attributes. More on the domain modelling for this medical imaging product family can be found in [1].

Product Family Architecture

A product family architecture has been set up based on the identified domain concepts and the scoping of the domain. Architectural styles and domain-specific architectures

(see page 7 of [5]) have been used within this product family architecture, taking the various quality attributes into account. The main architectural styles applied are:

- Layering
  The family architecture defines a decomposition into a number of independent layers. A layer (or parts of a layer) can be replaced independently of the others, providing support for diversity.
- Independence of units
  The family architecture defines a decomposition into a number of units. A unit contains a coherent set of functionalities, and covers a sub-domain of the overall 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. This is supported by letting each unit deal with its own relevant software aspects (e.g. error handling, initialisation, etc.), using events as notification mechanism, using blackboard-like information exchange mechanisms, etc.

In addition, domain-specific architectures (similar to architectural styles, but domain-specific) are applied. Within the medical imaging domain, the image acquisition chain plays an important role and can be captured in a domain specific architecture. This chain contains the various peripheral devices, ranging from the generation of radiation to the processing and storage of images. The synchronisation between these devices is not done by the devices themselves, but by higher-level control functionality. In software this leads to the identification of a number of units, each providing an abstraction of the underlying hardware device, e.g. the radiation generator or the image processor. These units are located in the technical layer, and are independent from each other. This is shown in figure 1, in which the decomposition into units is schematically depicted. The workflow using (and controlling) these devices comprises a number of phases, like patient administration, image acquisition, image viewing, image handling (including printing, archiving, communication via a network); see [13]. This leads to a domain-specific architecture with units, each representing a phase in the workflow. Since these units contain the application knowledge, they a located in the so-called application layer. In principle, these application units do not communicate directly with each other. Instead, blackboard-like mechanisms are used. Next to these two layers, an infrastructure layer is added providing basic facilities to the other two layers, like logging of events and watchdog-like facilities.

According to [12], in which different styles for generic architectures are discussed, the architecture schematically shown in figure 1 can be classified as a generic variance-free architecture. This means that, at this level, the variance has not yet been made explicit. The units have been identified and responsibilities have been assigned to them. Each unit is in principle relevant in the context of each family member, although some units are optional. Each unit must internally handle the family's diversity and consists of one or more software components that can be deployed separately [14]. So, one could say that there are two levels of (de)composition: the family architecture describes a first level decomposition into units, and each unit consists of one or more components. These components may be plug-ins of component frameworks, as will be discussed later on.

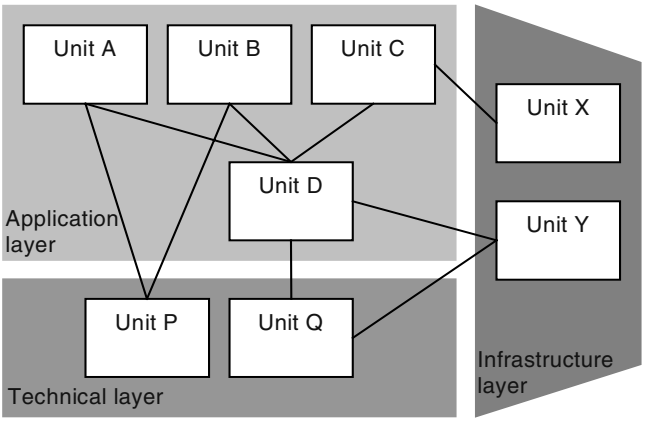


Fig. 1. Unit View of the Product Family Architecture

3. Handling Diversity

One of the most characteristic quality attributes in a product family is support of diversity. This section first describes the diversity in the medical imaging product family, followed by an example. After that it will be explained which mechanisms were used to realise this diversity and how these mechanisms fit in an architectural context.

Diversity of the Medical Imaging Product Family

A consequence of the diversity amongst the various family members is that the family architecture must support a variety of functions at the same time, which requires configurability. Furthermore, the family must be extensible over time. Diversity within a product family can stem from several sources. The main sources that can be identified are changes in the set of features that must be supported by the product family (relating to user requirements; see also [9] concerning the feature-oriented approach) and changes in the technologies that are used for the realisation, e.g. new hardware. When changes in the realisation are noticeable to end-users, they can also be regarded as changes in features. In our product family, examples of diversity in features are different procedures for acquiring images, different ways of analysing images, etc. Examples of hardware-related diversity are new implementations of image processing hardware, larger disks for image storage, and new devices for moving patients.

In the domain and design models, diversity can be specified in a number of ways, as illustrated in figure 2 for the functionality relating to the geometry[1] unit, one of the

[1] The geometry controls the main moving parts in the medical imaging family and determines the part of the patient to be examined and its projection on the image.

sub-domains in the medical imaging product family. One way of describing diversity is by allowing a range in the multiplicity of a certain concept; e.g. a Geometry may have one or more ImageDetectors. Another way of allowing for diversity is by using specialisations of generic concepts; e.g. a ModularGeometry is a special case of a Geometry. It is also possible to define generic concepts that are not further detailed on class level, but on object level, e.g. various instances of DefaultPositionMovements, each with their own behaviour.

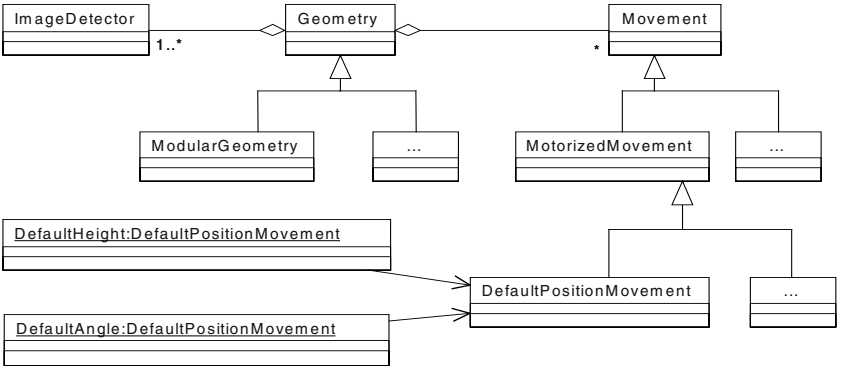


Fig. 2. Diversity in a UML Model of the Geometry Sub-domain

Realising Diversity

The architect has defined two principles for the family architecture that are relevant when selecting techniques for realising diversity; see [13]:

- binary reuse of components
- division of the product family development into a generic part and member-specific parts

Component frameworks support these two principles, as will be explained below. Generally, a framework forms a skeleton of an application that can be customised by an application developer. So, a framework supports the division of functionality into a generic skeleton part and a specific customisation part, covering the second principle.

Generally speaking, two types of frameworks exist: component frameworks and class frameworks. Component frameworks support extensibility by defining interfaces for components that can be plugged into the framework, enabling the binary reuse principle. Component frameworks are usually easier to use because of this principle, since class frameworks also use mechanisms like inheritance, and application developers require information on the internal framework structure (see also [4]). This ease of use is especially important in a situation in which component frameworks are applied at several distributed development sites. Szyperski ([14], page 26) describes a component framework as “a set of interfaces and rules for interaction that govern how components plugged into the framework may interact”.

Of course, not all diversity can be dealt with via component frameworks. Another mechanism used for realising diversity is based on configuration via data. Component frameworks are applied if significantly different functionalities are needed in different family members, e.g. a different geometry offering different movements. The configuration data is used when the differences are smaller, e.g. for country-specific settings.

Component Frameworks and Architecture

The product family architecture presented in section 0 specifies only a high-level view and applies to all the 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 software components and may contain variation points at which variation occurs for the various family members.

Component frameworks are applied as a means for supporting diversity. They support the definition of a family platform consisting of those software components[2] that are relevant for several (but not necessarily all) family members. In addition, a generic family skeleton [7] has been defined that is formed by those software components that are relevant for all the family members. This family skeleton is based on the unit structure and the generic components within the units. It is possible to construct such a family skeleton because the various subsystem domains are relevant for all the family members and the (expected) variation in these sub-domains is limited. Each unit may contain one or more component frameworks. The family skeleton is schematically shown in figure 3. Here, the unit relations and the dark grey parts (including component frameworks) form the family 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 graphically shown). More information on component frameworks and architecture is to be found in [15].

4. Kinds of Component Frameworks

A component framework defines one or more roles for participating components (plug-ins), relative to the framework, using contracts. The medical imaging family architecture comprises two kinds of component frameworks, which will be described in the next two paragraphs. The main differences between these two kinds concern whether the component framework is a software component itself managing the interaction of the plug-ins, and what kind of abstractions can be offered on the interfaces of the plug-ins.

[2] Besides the software, the platform also comprises requirements and design documentation, hardware, interface specifications, architectural rules and guidelines, tools, test environments, etc.

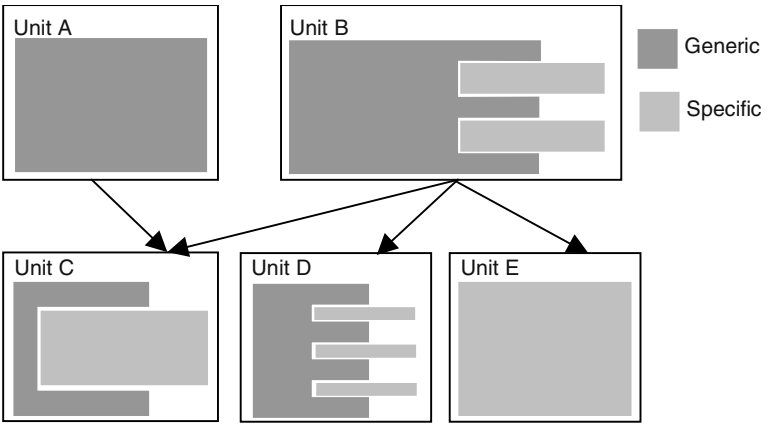


Fig. 3. Family Skeleton and Specific Components

The high-level family architecture defines units and their interfaces. These units can again be decomposed into a number of components, again with clearly defined interfaces. This leads to nested frameworks into which components, plug-ins, must be inserted that adhere to the defined interfaces (some components may be optional). Figure 4 gives an example of a framework (the grey part) that defines placeholders for three plug-ins. Such a component framework is not a software component itself, and plays no active role in connecting the plug-ins during initialisation-time or run-time; it only defines a co-operation structure for a fixed number of plug-ins via contracts based on interfaces. In this paper this is called a *structure component framework* (although dynamic issues are of course also relevant here).

The second kind of component framework is a software component itself, which will be called the *framework component*, and defines one or more roles to which one or more components can be plugged-in. Such a role defines some kind of *service* that the plug-ins provide via their interfaces. Such a service may range from a software representation of a scarce hardware resource, e.g. image processing nodes which provide services like noise reduction or image subtraction, to some piece of logical functionality, e.g. a spelling checker. This means that a plug-in is a container of one or more services. In addition to the plug-ins, there are usually clients that use the functionality provided by the component framework and plug-ins as a whole. Such a component framework contains a framework component, and this component is active in connecting interfaces of plug-ins and providing the functionalities, the services, of these interfaces to other clients; see figure 5. In this paper this will be called a *service component framework*.

Such a service component framework has been developed for example for the geometry sub-domain introduced in section 0. In the family architecture, geometry has been identified as a unit. This unit as a whole provides amongst others a number of movements for positioning a patient, like TableHeight and TiltTable. Various geometry hardware modules exist, each providing a subset of all the possible movements (about 100 movements in total). To deal with this diversity, the hardware model is mirrored onto software by one framework component, which provides the generic functionality, and a number of plug-ins, each providing a number of

movements to the framework component. In this example, the movements are thus modelled as the services that are provided by the plug-ins, related to the specific role. The various clients of this component framework can use these movements without worrying about the internal component framework structure. Next to the movements, also other kinds of services have been identified for the geometry unit for which other roles have been defined. This kind of framework has also been applied to other sub-domains of the product family, e.g. in acquisition for adding acquisition procedures, in reviewing for adding analytical functions, and in the field-service domain for field-service functions (calibration, configuration, etc.).

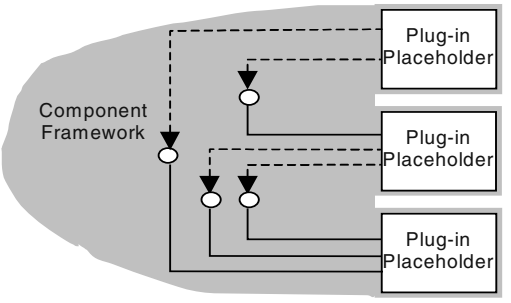


Fig. 4. Structure Component Framework defining a Co-operation Structure for Plug-ins

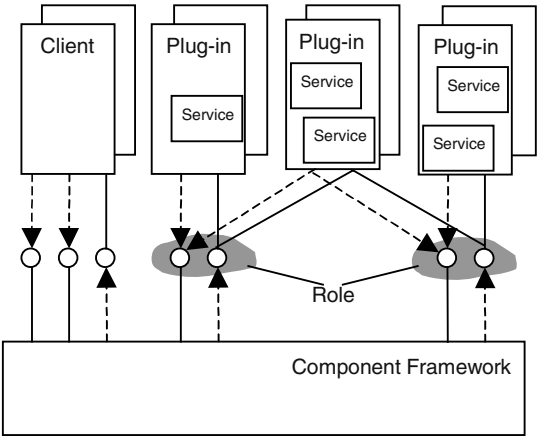


Fig. 5. Service Component Framework actively connecting Plug-ins and Clients

The difference between the structure and service component frameworks is related to the degree in which the functionality of the sub-domain involved can be standardised and modelled as services, and the support for diversity that is needed. In the geometry example the movements are identified as an important concept (see figure 2), and they are handled as services provided by plug-ins that match the hardware modules. The other kind of framework is applicable when less standardisation is possible in a certain sub-domain, and diversity can be supported by

replacing a complete software component. For example, the film detector unit (responsible for the positioning of the film), which, like the geometry unit, is a technical unit, exists in a number of variants. Support of this variation is achieved by replacing the complete hardware module. Furthermore, since the similarities between them are limited, only the interface of the unit is standardised, not the internal implementation. This variation is supported in software by replacing a complete software component that realises the functionality of the entire unit.

The remainder of this paper will focus on the service component frameworks because they are more specific and dedicated than other frameworks, allowing a discussion of common issues.

5. Service Component Frameworks

Two important issues concerning the service component frameworks are the responsibilities of the framework component, and the abstractions and interfaces that the plug-ins must provide. These two issues will be discussed in the next two subsections, followed by some observations on packaging component frameworks and plug-ins.

Framework Component and Plug-in Responsibilities

In defining a service component framework, a number of responsibilities must be taken into account for the framework component, viz.:

- Providing support for diversity
  The most important responsibility of the framework component is to support diversity. This means that the framework component must support easy pluggability. To this end, the (expected) forms of diversity must be carefully analysed and the right concepts must be selected. In the geometry example this is amongst others the movement concept. This is closely related to the interface issues discussed below.
- Providing a central access point
  Since it enables diversity and because this diversity should not propagate through the entire system, a framework component has to serve as a central access point for the functionality provided by the plug-ins without revealing the specific configuration of the plug-ins to the clients of this functionality. An important activity here is maintaining a list of all the available services provided by the plug-ins. In the geometry example, the movements are identified via a character string, describing the kind of movement, e.g. TableHeight, TiltTable, etc.
- Providing a basic infrastructure
  The plug-in should not interact directly with parts of the system that may vary over time. Instead, the framework component has to provide some kind of infrastructure to the plug-ins. This way, when the environment changes, only the framework component needs to be updated. Such an infrastructure is for example responsible for correctly initialising all the plug-ins and their services.

- Providing additional functionality
  In addition to the basic responsibilities mentioned above (connection management and infrastructure), the framework component may also contain additional functionality, since it is a component of its own. This is for example the case with the geometry unit in which the framework component is responsible for adding resource management and scheduling in order to deal with the scare movements in a controlled way. Another possibility is that the framework component already contains some services and can operate on its own, so that adding plug-ins to that framework is not mandatory.

The main responsibility of the plug-ins is to provide the functionality (services) to the framework component. This may take place during initialisation time or during run time when the service is actually needed. The architectural concepts, interfaces and infrastructure to which the plug-ins must adhere are defined by the component framework.

Interfaces

Since component frameworks have been introduced to deal with diversity, they must be able to deal with various plug-ins providing their own specific functionality. This functionality must be handled by the component framework in a generic way and provided to interested clients. This means that the right abstractions must be chosen for the interfaces between the plug-ins and the component framework and between the component framework and its clients.

Interfaces may be required to be clear and straightforward in use, and on the other hand they may be required to be stable. These requirements do not always agree with each other. Take for example the following interface (described in IDL):

```
interface IStableInterface : IUnknown
{
   HRESULT GetParameter([in] long ParID,
                        [out] long* ParValue);
   HRESULT SetParameter([in] long ParID,
                        [in] long ParValue);
   HRESULT PerformAction([in] long ActionID);
}
```

This overly generic interface, which is not based on any specific domain concept, can be used to provide any type of functionality. The advantage of this interface is that it will remain stable, even if new actions or parameters are added. The disadvantage is that it is no longer clear what kinds of actions are supported, since it does not refer to specific concepts. This may lead to incorrect use of the interface, requiring additional testing effort. So, one could say that in this case the syntax is stable, but the semantics are not, because the interface can be used for many different purposes.

We chose several interfaces for the movements in the geometry example, each representing one sub-group of movements, making the interface more specific and

easier to use. In this case the semantics are stable (a movement is a known concept in the domain), but the syntax may change slightly in the future.

```
interface IMovement : IUnknown
{
   HRESULT GetPositionValue([out] long* PositionValue);
   HRESULT GetSpeedValue([out] long* SpeedValue);
   …
}
```

Packaging Component Frameworks

When defining a unit or a component, a deliverable is provided to support the clients using this unit or component. This deliverable is called the requirements specification. A requirements specification specifies the interface that is provided to the clients. It includes the following parts:

- class diagrams
  These diagrams contain the interface classes that the unit/component provides to its clients. The classes in a class diagram are related to each other via associations, aggregations and generalisations. Each diagram has an annotation, describing the group of classes as a whole.
- sequence diagrams
  A sequence diagram represents an interaction, which is a set of messages exchanged between objects within a collaboration to effect a desired operation or result. In this context, the sequence deals with the interaction between the unit/component and its users. A sequence diagram contains a possible sequence of events, and is thus not complete in that sense.
- class descriptions
  A description is made for each class. This description is based on the model laid down by the class diagrams. The attributes, operations and state diagram of each class can be specified.
- software aspects
  Special attention is paid to the various software aspects identified by the architect that cut across most software components, like initialisation, error handling, graceful degradation, etc. These aspects are related to the quality attributes; see [10]. A separate section is identified for each relevant aspect.

Some additional issues must be taken into account when dealing with component frameworks. For example, the geometry unit consists of a service component framework with a number of plug-ins. In this case, the functionality provided by the geometry unit as a whole in a specific family member cannot be given by one document. Instead, two document types are used, viz.:

- generic requirements specification
  This requirements specification describes the generic interfaces that are provided to the clients. All the service concepts, in this case the various types of movements, their attributes, etc., are described in this document. However, no specific services (instances) are described.

- specific requirements specification
  Each plug-in provides its own specific requirements specification. This requirements specification describes the services provided and the plug-in's specific properties. The MaximumPosition, MaximumSpeed and other requirements need to be specified for each movement. The generic requirements specification focuses on the generic meaning of the service interfaces, whereas the specific requirements specifications deal with service-specific issues.

Another important aspect besides the pluggability of the actual software component plug-ins is the availability of this pluggability on a document level. This makes it easier to determine the total requirements specification for a specific family member, since each plug-in has a related document describing the specific services of that plug-in. This agrees with the idea that a component is the unit of packaging, i.e. a component consists not only of executable code, but also of its interface specifications, its test specifications, etc. (see also [3]).

A component framework defines the boundary conditions to which each plug-in must adhere. Since several plug-ins are usually developed for a framework, it is worthwhile to provide support for plug-in development, including:

- A documentation template can be provided for the specific requirements specifications. This will ensure that the author knows what needs to be specified and that all relevant issues are addressed.
- The interfaces and design of a component framework dictates the design of the plug-ins. That is why support can also be given for the design documentation of plug-ins.
- The interface files of the interfaces that the plug-in needs to support must be provided. Furthermore, some classes that are relevant for the design of each plug-in can be provided. It is also possible to provide a complete example plug-in, containing all the information relevant to a plug-in implementer.
- A test specification can be provided against which each plug-in must be tested. It is even possible to provide some test harness in which these tests can be performed automatically.

6. From Domain Model to Component Frameworks

As stated above, the product family architecture does not show diversity on the level of units. This means that the units form a stable family skeleton. The diversity can be found inside the units. One of the ways of dealing with this is to divide the unit model into a generic part (related to one or more component frameworks) and a specific part (the plug-ins), which is similar to the approach discussed in [8]. Each unit has a part of the overall domain model assigned to it as a starting point for the design activities. A number of steps in this iterative modelling process can be identified (using the geometry example):

1. The design activity adds design classes to the assigned domain model, e.g. a manager class maintaining a list of all the movements is introduced, and a priority-based scheduler class is added to handle the scarce movement resources.

2. Then the commonality and diversity are analysed. This analysis is closely related to the diversity in features and realisation techniques that the product family must support. Identify which part of the model will remain stable and which will vary from one family member to another and through time. The larger the selected generic part, the lesser the resulting flexibility.
3. On the basis of the previous analysis it must be identified which concepts must be used for the services that the plug-ins should provide to the component framework. In the example, the movements were selected as the services. It may be possible to handle the variation more efficiently by introducing generic concepts (these concepts may even be useful in the domain model). For example, various specific moving parts of a geometry have specific position information. By introducing a concept like a GeometryComponent class containing specific position information, this information can be handled in a generic way.
4. Identify which additional design concepts are needed by splitting the model into two parts, i.e. the framework component and the plug-ins. This splitting has more consequences for component frameworks than for example for class frameworks, in which techniques like inheritance may also be used. For example, the generic part is made responsible for starting up the plug-ins during the system's initialisation .

To summarise, step 1 contains the 'normal' design activity. In step 2, the diversity is analysed. The relevant service concepts that must be provided by the plug-ins are identified in step 3. Finally, step 4 introduces the infrastructure in which these services are handled.

Note that it is very difficult to obtain a completely stable component framework the first time. Many factors may lead to modifications in the interface, e.g. changes in the domain model, the choice of wrong concepts, etc. So, usually a couple of iterations will be needed.

7. Concluding Remarks

In this paper the component framework engineering has been described as part of the software development process for a medical imaging product family, focusing on service component frameworks. The purpose of introducing component frameworks is to support diversity. The quoted geometry unit example is one of several service component frameworks that have been introduced in the medical imaging product family.

The two important starting points for component framework engineering are the domain model, which provides the relevant domain concepts, and the product family architecture, which provides the main decomposition into units. Some units must deal with internal diversity. Such diversity can be handled by one or more service component frameworks per unit. To this end, the functionality must be split into a part that is realised in the framework component and a part that must be provided by the plug-ins.

The approach described in this paper is currently being applied at Philips Medical Systems, involving development groups distributed across various sites. A large part of the development and testing can be shared for the various family members.

Furthermore, the component frameworks define clear interfaces for extension with specific features. This approach is very promising as far as the first release of the family members is concerned. Of course, further releases must validate this approach.

The approach taken here is not specific to the medical domain. In fact, [7] describes a product family approach with related principles that has been very successfully applied for a telecommunication switching system family. Whether this approach is applicable will depend on the domain and requirements of the product family concerned. For example, it is relevant that the domain in which the family is positioned is relatively stable, which is the case in this mature medical imaging domain. The domain and its variation can then be modelled without having to expect fundamental changes. Furthermore, the individual members of the family must have sufficient commonalities in the sense that the sub-domains identified will be relevant for most of the family members (unlike in the approach described in [11], which deals with product populations), so that a stable family skeleton can be obtained for the development of each family member. The variation within each of the sub-domains must be bounded so that the relevant diversity can be modelled by (service) component frameworks. Finally, this approach is based on component-based development and is applicable to complex software-intensive product families, it allows a high degree of variation in the features supported by the individual family members via plug-ins and configuration parameters, it reuses a large part of the development and test effort by establishing a family platform, and it supports development in distributed development groups.

Acknowledgements

This research has been partially funded by ESAPS, project 99005 in ITEA, the Eureka Σ! 2023 Programme.

I would like to thank the chief architect Ben Pronk and all the people involved in the medical imaging product family development. I also thank my colleagues Pierre America, Jürgen Müller, Gerrit Muller, Rob van Ommering, Tobias Rötschke, and Marc Stroucken for comments on earlier versions of this paper.

References

[1] Pierre America, Jan van Wijgerden, *Requirements Modeling for Families of Complex Systems,* Proceedings of the IW-SAPF-3 (this volume), Las Palmas de Gran Canaria, March 2000.

[2] Len Bass, Paul Clements, Rick Kazman, *Software Architecture in Practice*, Addison Wesley, Reading, Mass., 1998.

[3] Desmond F. D'Souza, Alan C. Wills, *Objects, Components, and Frameworks with UML: The Catalysis Approach*, Addison Wesley, Reading, Mass., 1998.

[4] Mohamed E. Fayad, Douglas C. Schmidt, *Object-Oriented Application Frameworks*, Communications of the ACM, Vol. 40, No. 10, pages 32-38, October 1997.

[5] Christine Hofmeister, Robert Nord, Dilip Soni, *Applied Software Architecture*, Addison-Wesley, Reading, Mass., 1999.

[6] Ivar Jacobson, Martin Griss, Patrick Jonsson, *Software Reuse – Architecture, Process and Organization for Business Success*, Addison Wesley, Reading, Mass., 1997.

[7] Frank J. van der Linden, Jürgen K. Müller, *Creating Architectures with Building Blocks*, IEEE Software Vol. 12, No. 6, pages 51-60, November 1995.

[8] Jacques Meekel, Thomas B. Horton, Charlie Mellone, *Architecting for Domain Variability*, Proceedings of the Second International ESPRIT ARES Workshop, pages 205-213, Springer Verlag, Berlin Heidelberg, 1998.

[9] Jürgen K. Müller, *Feature-Oriented Software Structuring*, Proceedings of COMPSAC '97, pages 552-555, August 1997.

[10] Jürgen K. Müller, *Aspect Design with the Building Block Method*, Proceedings of the First Working IFIP Conference on Software Architecture, pages 585 – 601, Kluwer Academic Publishers, 1999.

[11] Rob van Ommering, *Beyond Product Families: Building a Product Population?*, Proceedings of the IW-SAPF-3 (this volume), Las Palmas de Gran Canaria, March 2000.

[12] Dewayne E. Perry, *Generic Architecture Descriptions for Product Lines*, Proceedings of the Second International ESPRIT ARES Workshop, pages 51-56, Springer Verlag, Berlin Heidelberg, 1998.

[13] Ben J. Pronk, *Medical Product Line Architectures – 12 years of experience*, Proceedings of the First Working IFIP Conference on Software Architecture, pages 357 – 367, Kluwer Academic Publishers, 1999.

[14] Clemens Szyperski, *Component Software – Beyond Object-Oriented Programming*, Addison Wesley, Reading, Mass., 1997.

[15] Jan Gerben Wijnstra, *Supporting Diversity with Component Frameworks as Architectural Elements*, to appear in the Proceedings of the ICSE 2000, Limerick, June 2000.

Meeting the Product Line Goals for an Embedded Real-Time System

Robert L. Nord

Siemens Corporate Research
755 College Road East
Princeton, NJ 08540 USA
rnord@scr.siemens.com

Abstract. This paper describes the software architecture design of a real-time monitoring system. The system has different configurations to produce a set of medium to high-end monitoring systems. The system was designed to meet the product line quality objectives of configurability, extensibility, and portability within the constraints of an embedded real-time system. These objectives were achieved using global analysis and the principle of separation of concerns to organize the architecture into multiple views. The major design artifacts in each of the views were: (1) a publish-subscribe communication protocol among software entities that allowed requirements to be isolated to a component; (2) a layered design that provided encapsulation of common services; (3) and a task structure that facilitated flexible process allocation.

Keywords: Software engineering, software architecture, product line, global analysis, multiple views, design decisions, industrial applications.

1 Introduction

This paper describes the architecture of an embedded, real-time patient monitoring system. It is a stand-alone bedside unit that obtains and displays a patient's vital signs, or sends them to a central unit for display. The bedside unit can be transported along with a patient, so physical size and cost limitations impose severe hardware constraints. The system has different configurations to produce a product line of medium to high-end patient monitors. The system was designed to meet the product line quality objectives of configurability, extensibility, and portability within the constraints of an embedded real-time system.

The architecture was designed to be configurable. The architecture serves as the basis for medium to high-end products. Customer setups have a lot of commonality but many minor differences as well. The user is able to customize the system with hundreds of user settings. Different levels of functionality are provided in the same software versions based upon user purchased options. The packaging of the software can be easily changed in response to market conditions. The system supports hundreds of

F. van der Linden (Ed.): IW-SAPF-3, LNCS 1951, pp. 19-29, 2000.

parameters through directly connected sensors and through third party devices via a standard communication interface.

The architecture was designed be extensible. Through a phased release, new features are constantly added. The monitor supports thousands of application rules. These rules are dynamic in nature and are likely to become more complex in the future driven by changes in hardware, software, and the application domain. Major new features are constantly added to the system usually with little or no modification to the rest of the system. Minor requirement changes can usually be handled by changing a single software component. Adding new features or implementing requirement changes did not affect the integrity of other already working features.

The architecture was designed to be portable. Changes in the hardware and software platforms were anticipated in order to take advantage of new technology that provides increasing power at lower costs. The system runs on three operating systems, communicates on two networks, uses three different graphics libraries, and executes on two different target-hardware platforms (one of which has a single main processor and the other has two main processors). It was easy to move the product to each of the new environments. Porting to a new operating system only affected the inter-process communication libraries and implementation of the operating system library interfaces. Moving to a different graphics library only required a rewrite of the implementation of the generic graphics library.

The architecture was designed to satisfy the real-time performance requirements. The monitor must provide correct and timely information to the user. Deadlines and priority levels could change based on changes to requirements or the platform. There is flexibility in assigning modules to processes allowing for tuning to meet these requirements. This was facilitated by limiting interactions between modules. Local optimizations were employed (e.g., minimize data copying, produce data only when needed). These maintained the integrity of the architecture and were transparent to the software components. Because of the amount of data and the time critical requirements, displaying waveforms on the screen necessitated a special transfer mechanism, opting out of the publish-subscribe communication pattern defined by the architecture. The publish-subscribe protocol served the performance needs for the rest of the system.

Successful architects analyze factors that have a global influence on the system. This global analysis task begins, as the architecture is defined [2,4]. Its purpose is to capture the factors that influence the architecture design, characterizing how they change. Global analysis takes on an even more prominent role in product line design. The architect must characterize how the influencing factors vary among the products within a product line. The architect develops and selects strategies in response to these factors to make global decisions about the architecture that allows the developers of the products to make uniform decisions locally. Guiding the developers in this way ensures the integrity of the architecture. This is an iterative process. During the design, certain decisions feed back into the global analysis, resulting in new strategies.

The remaining sections describe the system and the global analysis activities that led to the architectural decisions for meeting the product line goals.

2 System Overview

The primary function of the system is to acquire data and display it to the user. This functionality is seen in the scenario depicted in Figure 1. Scenarios are used to show how data flows among the major functional components of the system.

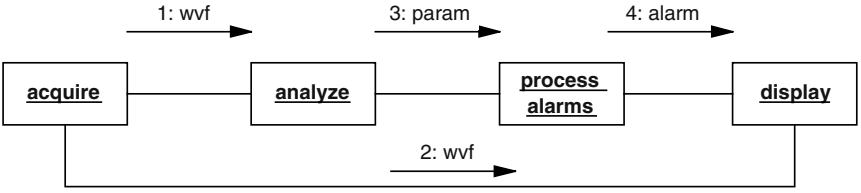


Fig. 1. Producer Consumer Processing Scenario

The scenario is described by a UML object interaction diagram that shows how raw data is acquired, processed, and displayed on the screen [5]. Arrows, labeled with a sequence number and the type of data indicate data flow between entities. The waveform data is first acquired. The acquired data is both displayed on the screen and processed to produce the raw parameter values. These values are further processed to produce the alarm information for the clients and displayed on the screen.

In addition to these functional requirements, the architecture design was influenced by the need of the monitoring system to be configurable, handle a flexible and growing set of requirements, run on different hardware and software platforms, and meet high availability, real-time performance, and quality assurance requirements.

These objectives and the influence they exerted on the architecture are summarized in Table 1.

Table 1. Quality Objectives

Objective	How Achieved
Configurability	Isolate requirement to a component Device management environment
Extensibility	Rule-based system Isolate requirement to a component Separate policy from procedure
Portability	Layered design Software entity template Common processing libraries
Real-time Performance	Flexible task allocation

Global analysis and architecture design activities are structured according to four architecture views [7]. Each of the four views describes particular aspects of the sys-

tem. The resulting architecture centered on the following design artifacts in three of the views:

- a publish-subscribe communication protocol among software entities that allowed requirements to be isolated to a component,
- a layered system that provided encapsulation of common services,
- and a task structure that facilitated flexible process allocation.

The following section provides the rationale for how the architecture resulted from the objectives.

3 Global Analysis

For the monitoring system, there are a multitude of factors that influenced the architecture. Some of the factors the architect considered included the skills of the developers, the schedule, general-purpose and domain-specific hardware, product features, and performance. In order to bring some order to these factors, they can be organized around the quality objectives that the architecture must meet. Within each of the objectives there will be a number of issues to address. Here we will focus on three issues that have a significant impact on the architecture: adding and configuring features, changing the software platform, and meeting the latency requirements.

Issue: Easy Addition and Configuration of Features

There are many application-policy rules that are dynamic in nature. These application rules place requirements on the software components with respect to what action they must perform, what information they need, and what information they must produce. They are likely to become more complex in the future driven by changes in hardware, software, and the signal-processing application domain.

The monitoring system must dynamically adapt to what features are presented to the user depending on what network and I/O devices it is connected to. Designing a system for easy addition of features is complicated by the requirement that these changes can occur while the monitor is running.

Influencing Factors

- There are hundreds of customized settings and thousands of applications rules. New settings and rules are being added regularly. This has a large impact on all of the components.
- The monitor can be moved to a different network. It must handle dynamically changing features based on the network type. This affects the user interface.
- The monitor must communicate with third party devices and dynamically adapt to the devices connected to it. New devices could be added in the future. This affects the data acquisition components.

- Different levels of functionality can be provided depending on user purchases. The packaging of software is likely to change in response to market conditions. This has a moderate impact on all components.

Solution

The application rules defined in the requirements suggest a rule-based solution. While this approach facilitates modifiability, there were concerns about performance. The data flow processing nature of the acquisition suggests a pipes and filters solution.

The solution adopted was to build applications using loosely coupled software components that communicate using a publish-subscribe protocol.

- *Strategy: Separate monitor functionality into loosely coupled components*. Separate monitor functionality into software components that represent a logically related set of application features.
- *Strategy: Introduce publish-subscribe communication*. Each component owns or is responsible for publishing information that may be of interest to other components in the system. Other components are able to subscribe to the information and are informed of changes as they happen.

The solution incorporates aspects of both the pipes and filters and rule-based approach. In one sense the solution is like a bulletin board where components watch for data of interest, perform an action, and then update the global state. But it is not a general rule-based system since the data flow of high-level functionality can be identified – data is acquired, analyzed, and displayed. This helps identify end-to-end deadlines in order to meet the real-time requirements.

The architecture starts to emerge as a set of design constraints and decisions. The first such decision is to define the notion of an interacting software component and the abstract connections among them in the conceptual architecture view. Figure 2 is a UML class diagram that shows the structure of a software entity.

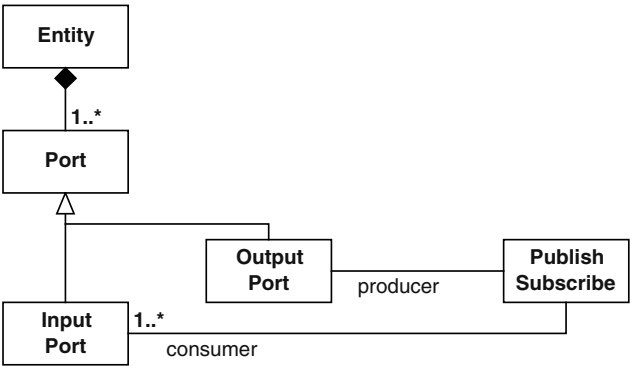


Fig. 2. Entity Structure

A software entity communicates with other entities through their ports via messages. There are different message types for the data handled by the monitor. These include the raw data coming in from the devices, waveform samples, parameter val-

ues, and alarm status. Ports are of two types. Entities publish information via its output ports and subscribe to information via its input ports. One entity is responsible for producing a specific message that one or more other entities consume. Entities are decoupled in the sense that producers don't know what entities are consuming its data and consumers do not know the entity responsible for producing the data.

The PublishSubscribe connector represents an abstract connection between a producer and a consumer. Entities first register for data of interest. When published data is written, it is disseminated to all entities subscribed to it.

Meeting the Objective

Providing a uniform model of loosely coupled components and explicitly recognizing the communication infrastructure provided developers with the flexibility to construct the varying product applications. It also allowed them to focus on the functionality of the components without regard to what other components were using it and how data would be transferred in order to meet the requirements for extensibility and configurability.

Issue: Changing the Software Platform

The software platform consisting of the operating system, communication mechanisms, and graphics commands is likely to change when the applications are ported to new hardware platforms.

Influencing Factors

- Portability requires that the system operate on different graphics environments, operating systems, and processors. These hardware and software platforms are likely to change. They may not provide the same services or have the same interfaces. This has a large impact on all of the components.
- Support is needed for client server applications in a distributed environment. Middleware support must be provided on top of the real-time operating system.

Solution

Additional infrastructure software was added between the product applications and the hardware platform; this included networking software, communication mechanisms and interfaces, database mechanisms and interface, and a timer interface.

- *Strategy: Encapsulate operating system and communication mechanisms*. Encapsulate operating system dependencies into an OS library and interprocess communication library which include software timer support.
- *Strategy: Encapsulate graphics*. Create generic graphics interface and encapsulate dependencies in a library so that display software can be easily ported to different graphics environments.

- *Strategy: Encapsulate data management*. Create a generic data management interface to a central repository that serves as the dominant mechanism by which applications share data.
- *Strategy: Encapsulate device management.* Create a generic device management interface that provides applications with a uniform way to access devices.

The application software is layered on top of the platform, graphics, and device manager software as seen in Figure 3. Layers are represented as UML packages. The layering structure from the module architecture view is used to implement the global analysis strategies of providing abstract interfaces and isolating dependencies. The software platform provides an interface to the host platform to enhance portability.

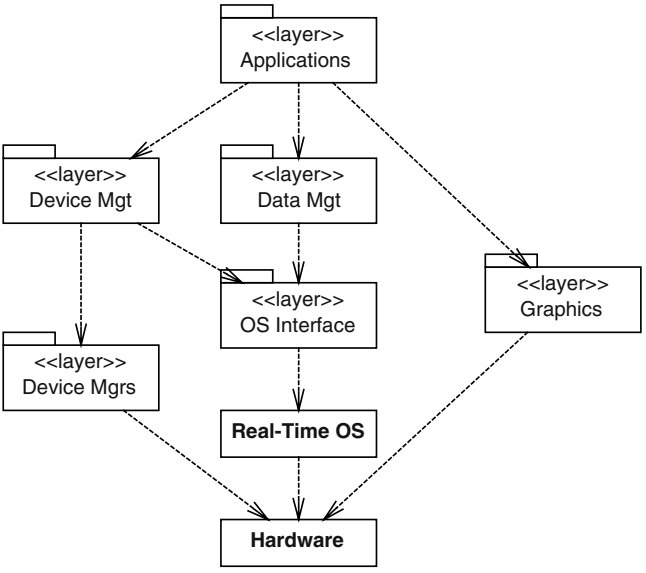


Fig. 3. Layering Structure

More details about the infrastructure for managing communication among entities emerge. There are modules corresponding to each entity in the conceptual architecture. These modules are organized in the Applications and Device Managers layers. The publish-subscribe connector is implemented by modules in the Data Management layer.

Data management is part of the software platform and was developed in response to the extensibility requirements to provide support for communication between entities. Portability places additional requirements on the solution regarding interfaces and middleware support. Data management is responsible for providing support for data publishing and registration, data distribution, and access to local and remote data.

Meeting the Objective

The collection of common application programmer interfaces (APIs) for device services, data management, graphics, and the operating system provided the solution for

meeting the portability requirements. These APIs accommodated the variations among the products such as the number and types of devices, processors, and operating systems. When changing the software platform, changes are isolated to the implementation of the graphics or OS interface, or to the implementation of a device driver adhering to the device manager API.

Issue: Meeting the Latency Requirements

Meeting the real-time performance requirements is critical to the success of the product. Data must be shown on the display as it is being acquired and alarms must be enunciated within milliseconds of detection.

Influencing Factors

- The processor speed is likely to change very frequently, even during development. As technology improves, it is desirable to be able to use faster processors or change the allocation of processors. There is no impact to the applications unless the operating system changes.
- Monitors must provide correct and timely information to the user. The maximum delays, specifying the latency of the information, are not negotiable. This has a large impact on data acquisition and processing components.
- Several different processing priority levels are required based on different processing deadlines. Deadlines and priority levels could change based on changes to the requirements or the platform.

Solution

Performance has been considered when coming up with the entity infrastructure. Decomposing the system into separate components provided the flexibility to allocate them to different processes. Entities were assigned resource budgets that provided a baseline for performance estimates that could be tracked and refined over time.

Multiple processors and a real-time operating system were also used to help meet the high performance requirements. Different processor configurations can't be ruled out in the future. This reinforces the need for the portability objective and the solution to encapsulate the software platform.

- *Strategy: Divide logical processing into multiple components*. Divide logical processing into multiple software components to meet the timing deadlines.
- *Strategy: Create a task for each unique processing deadline*. Use rate-monotonic analysis [3] as a guide to assign entities to tasks based on processing deadlines and to give priority to tasks which run more frequently or have a shorter deadline.
- *Strategy: Use multiple processors*. Choose a digital signal processor for signal filtering, use the fastest general-purpose processor available for main processing, and move graphics processing and waveform drawing to a graphics processor.

Tasks have a structure as shown in Figure 4. The structure is depicted using a UML class diagram. The heavy border around the task indicates that it is an active class. Its instances own a thread of control and can control activity. Each task has a

message queue that receives messages. The group control module contains the control logic software that routes the message to the appropriate entity.

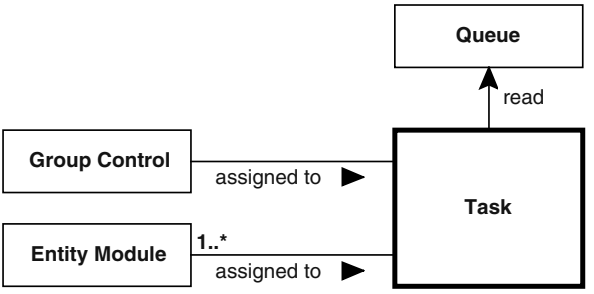


Fig. 4. Task Structure

Constraints are placed on the entities so that they can be flexibly allocated to processes. They must be non-blocking and communicate with each other by message passing only. The trade-off was adding complexity to the design of the modules to conform to this asynchronous access request model of interaction.

Entity modules that are defined in the module structure are assigned to a task based on processing deadlines following the principle of rate monotonic analysis (more stringent deadline tasks have higher priority). Entity modules with similar processing deadlines are placed in the same task; entity modules with unlike processing deadlines are placed in different tasks.

The data management module that implements the publish-subscribe connector is not a separate process but rather is implemented as a library that is linked into each of the tasks. This way the data manager does not become a bottleneck since messages are sent directly to subscribers. The trade-off was in distributing the overhead of disseminating messages to each of the entities.

It was recognized from the beginning that the publish-subscribe protocol would not be sufficient for handling the high speed data transfer necessary to display the waveforms on the screen in real-time. An optimized waveform connector was used for this special case.

Meeting the Objective

A uniform model of the task structure and guidelines for developing entity modules and assigning them to tasks provided the necessary constraints that the real-time performance requirements would be met. At the same time, local decisions could be made concerning the number of tasks, assignment of modules to tasks, priorities and runtime configuration in order to accommodate variations among the products and to provide the flexibility to tune the system to enhance performance.

4 Lessons Learned

Systems are growing in complexity. Understanding the entire product is a daunting proposition. At the time of inception, the need for a common product line architecture was recognized. A common architecture would be usable on multiple products, provide services independent of the operating system, user interface, hardware, and network, and promote the portability of the software. The technical goals were to carefully separate concerns to accommodate variations among products and to reduce the impact of change as the system evolved over time. Underlying these goals was the use of global analysis and multiple views.

Global analysis yielded a set of constraints on a collection of component types and their patterns of interaction for the product line. These building blocks were developed from first principles and the experience of building previous products. The constraints were enforced through design guidelines, training, software code templates, and the code architecture view. Component types, their relationships, properties, and constraints define an architectural style [1,6]. As experience grows these patterns may be recognized as styles and the architect could select common styles from a repository. The styles embody a set of predefined design decisions. Constraints that emerge during global analysis could be used to select the appropriate ones. In subsequent phases of design, the strategies are used to refine the previous design decisions and instantiate the component types in the construction of the architecture.

Multiple views is one way of using the principle of separating concerns to make building the system manageable. Previous products started the software architecture design with the execution view, which then dominated the architecture and limited the evolution and portability of the products. To support the system's planned configurations and additional ones in the future, the architects designed an explicit module view that was separate from the execution view. The system did not have an explicit conceptual view originally, although it was implicitly there. This use of multiple views (including an explicit code architecture view) proved critical in meeting the product line requirements of building an extensible, portable, and configurable system.

Architecture provides a framework for the developers.

- The entity structure in the conceptual view allows the entity designers to focus on the functionality of the components and promotes extensibility and configurability. Making the architecture infrastructure explicit provides a basis for reuse and ensures the integrity of the architecture.
- The layering structure in the module view provides a framework for hiding the details of the software platform from the applications and promotes portability. Common libraries provided a standard interface for the operating system, interprocess communication, device management, graphics, and database access.
- The task structure in the execution view provides the flexibility to tune the system to meet the performance objectives.

The architecture does not make the hard parts disappear but it makes them more manageable. Additional resources are needed to support the architecture. But this additional support results in the benefits of improving communication among the team and documenting the early design decisions.

References

1. Bass, L., Clements, P., and Kazman, R. *Software Architecture in Practice*, Reading, MA: Addison-Wesley, 1998.
2. Hofmeister, C., Nord, R., Soni, D. *Applied Software Architecture*, Reading, MA: Addison-Wesley, 2000.
3. Klein, M., Ralya, T., Pollak, B., Obenza, R., Gonzales Harbour, M. *A Practitioner's Handbook for Real-Time Analysis,* Boston: Kluwer Academic, 1993.
4. Nord, R.L., Hofmeister, C., Soni, D. Preparing for Change in the Architecture Design of Large Software Systems, Position paper accepted at the *TC2 First Working IFIP Conference on Software Architecture (WICSA1)*, 1999.
5. Rumbaugh, J., Jacobson, I., Booch, G. *The Unified Modeling Language Reference Manual*, Reading MA: Addison-Wesley, 1999.
6. Shaw, M. and Garlan, D. *Software Architecture: Perspectives on an Emerging Discipline*, Upper Saddle River, NJ: Prentice-Hall, 1996.
7. Soni, D., Nord, R.L., and Hofmeister, C. Software Architecture in Industrial Applications, in *Proceedings of the 17th International Conference on Software Engineering*, New York: ACM Press, 1995.

A Two-Part Architectural Model as Basis for Frequency Converter Product Families

Hans Peter Jepsen and Flemming Nielsen

Danfoss Drives, Ulsnaes 1, DK-6300 Graasten, Denmark
{hans_peter_jepsen,fl_nielsen}@danfoss.com

Abstract. Frequency converters, as well as other embedded systems in the process control and measurement domains, are designed to perform continuous processing of input values and to produce and maintain corresponding output values. At the same time these systems have to react on events that reconfigure the processing algorithms. This paper will present an architectural model that separates the event handling and the continuous data processing and let the event handling part select, which of multiple control or output algorithms in the continuous processing part has to be the active algorithm. Finally it presents, how this architectural model has given raise to concrete architectural elements.

Introduction

Danfoss Drives - one of the largest producers of frequency converters - is in a situation like many others: we have to produce a number of product series with an increasing number of variants, where we have to decrease time-to-market and keep development costs low. In order to meet these challenges, we are developing a product family software architecture, which is based on object-oriented principles and makes systematic software reuse possible.

Some of the first steps developing this architecture have been taken inside the Danish research project Centre for Object Technology (COT)[1], where inside a pilot project, an object-oriented model and a corresponding prototype implementation in C++ for the central parts of a VLT®[2] frequency converter were developed [4].

---

[1] The ***Centre for Object Technology (COT)*** is a Danish collaboration project with participants from industry, universities and technological institutes. The overall objective of Centre for Object Technology (COT) is to conduct research, development and technology transfer in the area of object-oriented software construction, for more details see [COT]. The activities in COT are based on the actual requirements of the participating industrial partners. COT is organised with six industrial cases, and Danfoss is participating in case 2 entitled "Architecture of embedded systems". The purpose of COT case 2 is to experiment with the introduction of object technology in embedded systems.

[2] VLT® is the trademark for frequency converters produced by Danfoss Drives.

F. van der Linden (Ed.): IW-SAPF-3, LNCS 1951, pp. 30-38, 2000.

A major breakthrough in the modelling of our frequency converter was the formulation of a conceptual model, which we call "the two-part architectural model". A major inspiration for this model was the process control architectural style from [2] along with the oscilloscope and cruise control examples from the same book.

Although the two-part architectural model may seem - and in fact is - very simple, it has been of great help since we formulated it. Further we think, that this model will be relevant for many other types of embedded systems, such as systems in the process control and measurement domains, which are designed to perform continuous processing of input values and to produce and maintain corresponding output values. At the same time these systems have to react on events that reconfigure the processing algorithms.

In this paper we will present the two-part architectural model, and show how this model relates to the architectural style "process control" [2]. Further we will present, how this architectural model has given raise to concrete architectural elements, and finally describe the benefits we have experienced, that this model gives us. The reader should be aware, that the background for the ideas presented here is only based on our work with frequency converters. We have not found a similar approach described elsewhere - neither on the frequency converter domain nor on other domains – although our searching has been limited.

Frequency Converters

A ***frequency converter*** is an electronic power conversion device used to control shaft speed or torque of a three-phase induction motor to match the need of a given application. This is done by controlling the frequency and corresponding voltage on the (three) motor cables.

A typical application - using a frequency converter in connection with a fan to maintain a desired room temperature - will illustrate the use of a frequency converter.

It is possible to save a lot of energy by reducing the fan speed as much as possible while maintaining the desired temperature. This is a normal closed-loop feedback control problem.

A problem, that often occurs when using a frequency converter in connection with a fan, is, that when the motor is not powered, the fan is "wind milling", i.e. running backwards. Therefore, before applying power, the fan must be stopped or "caught".

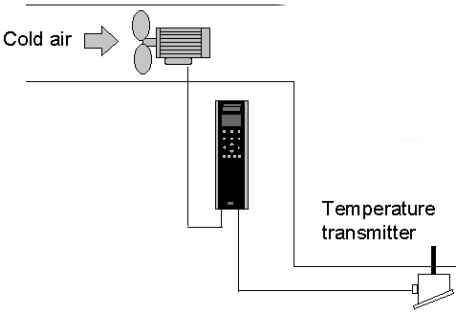


Fig. 1. A frequency converter used for controlling the speed of a fan.

The following scenario will give an idea of the working of the software, that this application requires:

Initially the motor is coasted (i.e. no output is applied to the motor) and the fan is wind milling. When the frequency converter receives a "start" command, it starts performing an algorithm, that we call "catch spinning motor". Whenever this

algorithm has detected the motor speed and adjusted its output to the motor, we have reached a situation where the frequency converter is controlling the motor. The algorithm sends an event with this information. The algorithm is then replaced with a "process closed loop" algorithm, which maintains the desired temperature.

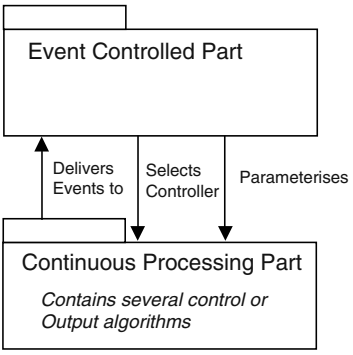
In the example above we see three modes of steady-state operation, namely "coast", "catch spinning motor" and "process closed loop". We also see, that the shift from one type of control to another happens as a result of an event, where the event can come from outside (the "start" command) or from the control algorithm ("spinning motor caught").

We want to stress, that this is only a simple example. Advanced frequency converters can be used in a large number of applications and have a high degree of complexity.[3] For our company's products the number of "modes of steady-state operation" is between ten and twenty, the number of the external events is at the same scale, and the number of internal events a bit higher. To adapt the frequency converter to the motor or the application, the user configures it, by applying values or choices to about 150 configuration items (called "parameters")

The Two-Part Architectural Model

As part of our first steps in developing the product family software architecture, that will be the basis for our future products, we have formulated a model, that we have called the "the two-part architectural model". Here is a very short description of this model:

- One part is the part of the system that is responsible for the continuous data processing, e.g. process control and measurement.
- The other part is the part of the system responsible for handling events - occurring asynchronously with the continuous data processing. (The configuration data for the system is also placed here since configuration changes is event controlled.)
- The event-controlled part configures (parameterises) the individual algorithms in the continuous data processing part, but also determines which one of the algorithms has to be the active control algorithm. On the other hand the continuous data processing part can produce events that have to be handled in the event-controlled part.



The continuous processing part contains the hardware and software controlling the motor in the "modes of steady-state operation". In the software case the processing is

[3] As an example the software for the VLT® 5000 series of frequency converter consists of approx. 150.000 lines of C-code. Beside a Motorola MC68331 micro controller the computing hardware design is based on an ASIC, which - measured by the number of gates - has a complexity as an 80386 CPU.

handled by periodically activated software, driven by periodic interrupts - since no real continuous (analogue) processing can happen in software.

Where we have tried to apply it, we have found, that it is fruitful to use the process-control model [2] to describe each of the "modes of steady-state operation". When we tried to make a use case model for our system, we had problems making use cases for the continuous processing parts of it. Later we found, that a description according to the process-control model actually could "play the role" of a use case.

Some of the control problems we have are simple open-loop problems, but most of them are closed-loop problems, either feedback or feed-forward.

Looking at the internals of the control algorithms (the controller part of each of the process control problems), we have found, that this algorithm normally has a simple dataflow structure, which can be described using the "Pipes and Filters" style.

A constraint, that the continuous processing part must handle, is that when replacing one control/output algorithm with another, there normally is a demand for "bumpless transfer", i.e. as smooth a shift as possible from one algorithm to another, in terms of disturbances on the motor shaft.

The event-controlled part is responsible for high-level motor control, i.e. how to safely bring the motor from one steady situation to another – both situations that the user recognises as steady – based on the events it receives. It is important to assure, that the system is safe, i.e. no obscure combination of events will make it unsafe.

To illustrate the complexity of this, we can mention, that in the system we currently have on the market the dominating event handling is described in a state-event matrix with approx. 60 states and 40 events.

Fan Control Revisited

To make it concrete we will illustrate the two-part model with the fan-control application described above.

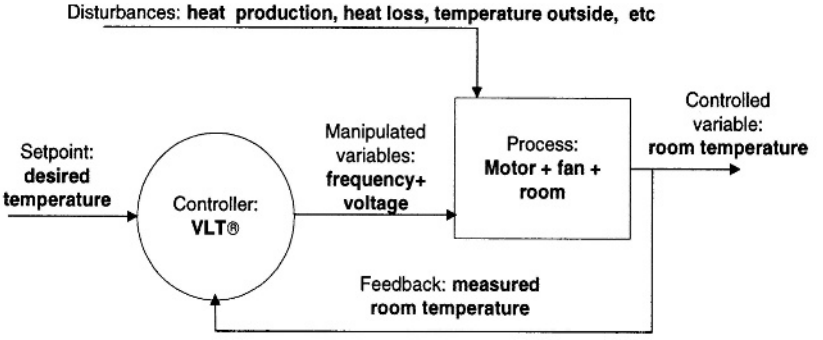


Fig. 2. Closed loop feedback process control of room temperature

The process-control model [2] - in this case a closed loop control – is shown with the relevant terms for this application.

The responsibility of the software required for this application, i.e. the internal structure of the "Controller: VLT" "bubble", is shown on the figure below as a block diagram where the data flow structure is easily recognised.

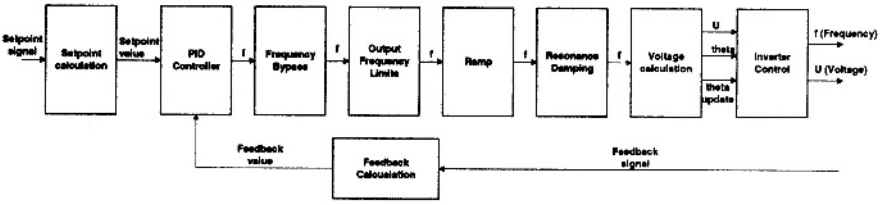


Fig. 3. Block diagram of the internals of the controller in fig. 2

Let us add a few comments to some of the processing blocks on figure 3. The four blocks "Frequency Bypass" to "Resonance Damping" do possibly change the desired output frequency, based on configuration values given by the user (e.g. "Ramp" will limit the changes (slope) in the output frequency to a value, set by the user). "Inverter Control" (implemented in a application specific chip (ASIC)) is a "continuous" production of a voltage vector, which has the desired amplitude and frequency.

For the "catch spinning motor" case, a diagram similar to figure 2 (a feed-forward diagram however) and also a block diagram similar to figure 3 can be made.

We are now ready to examine the two-part model introduced above by looking at a very simplified "picture" of the internals of a frequency converter.

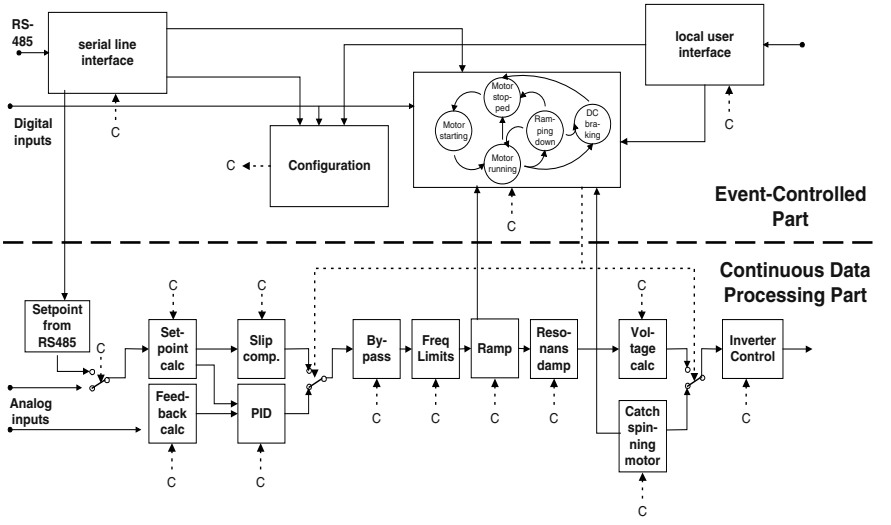


Fig. 4. Architecture view of frequency converter software - very simplified

The part *below* the dashed line represents the part of the system that is responsible for the continuous data processing, e.g. process control and measurement. In a given situation the data is flowing through a subset of the "blocks". The actual subset of blocks is given by certain setting of the "switches".

The part *above* the dashed line represents the part of the system responsible for handling events occurring asynchronously with the continuous data processing.

The event-controlled part configures (parameterises) the "blocks" in the continuous data processing part (represented with the arrows marked "C" to the blocks), but also determines the signal path in this part (represented with the arrows to the switches). On the other hand the continuous data processing part can produce events that have to be handled in the event-controlled part (represented with the up-arrows to the state-chart block).

To show the mapping of frequency converter-software required for the fan application onto the two-part architecture we will revisit the scenario above: Initially the motor is coasted and the fan is wind milling. When the event-controlled part of the frequency converter receives a start command from e.g. the digital inputs, it sets the switches in the continuous part to perform an algorithm, that we call "catch spinning motor". Whenever this algorithm has detected the motor speed and adjusted its output to the motor, we have reached a situation where the frequency converter is controlling the motor. The algorithm sends an event to the event-controlled part with this information. The event-controlled part then changes the settings of the continuous part to perform a "process closed loop" algorithm.

Is the Two-Part Model an Architectural Style – Rather Than a Model?

The two-part model described above builds on and encapsulates several architectural styles. The continuous processing part contains multiple continuous data processing algorithms, and in case they are control loop algorithms they can be described according to the *process control* architectural style. Further each of these algorithms can be divided in components that are connected according to *pipes and filters* style. Finally the event-controlled part encapsulates a *state-based control* style.

Why not rely just on the classical process control architectural style? The process control architectural style describes a single process control situation, and does this very well. However many systems must be able to handle a number of these process control situations, and when shifting between these, there often is a demand for "bumpless transfer". The two-part model tries to cover multiple controllers and bumpless transfer – handled by continuous processing part. Further "continuous data processing" does not have to be process control. Measurement is another example. Other types of continuous data processing are also covered.

The classical process control architectural style contains handling of discrete events. We have however chosen to place the handling of events external to the process control algorithms. The reason for this is that we believe, this gives us a better overview of the system.

A question that we have asked ourselves several times is: Is the two-part model really an architectural style? Is it possible to define "a vocabulary of components and connector types, and a set of constraints on how they can be combined [2]"?

Elements of the Concrete Architecture

Now we will describe, how we have realised some central parts of this model, namely

- how to have multiple process controllers or other continuous processing algorithms as part of continuous processing part,
- how the event-controlled part configures the continuous processing part and
- how bumpless transfer is handled.

First - inspired by the Strategy design pattern - the process controllers or other continuous processing algorithms are implemented as subclasses of an abstract class named `OutputController`. The actual algorithm behaviour is implemented by the method `generateOutput`.

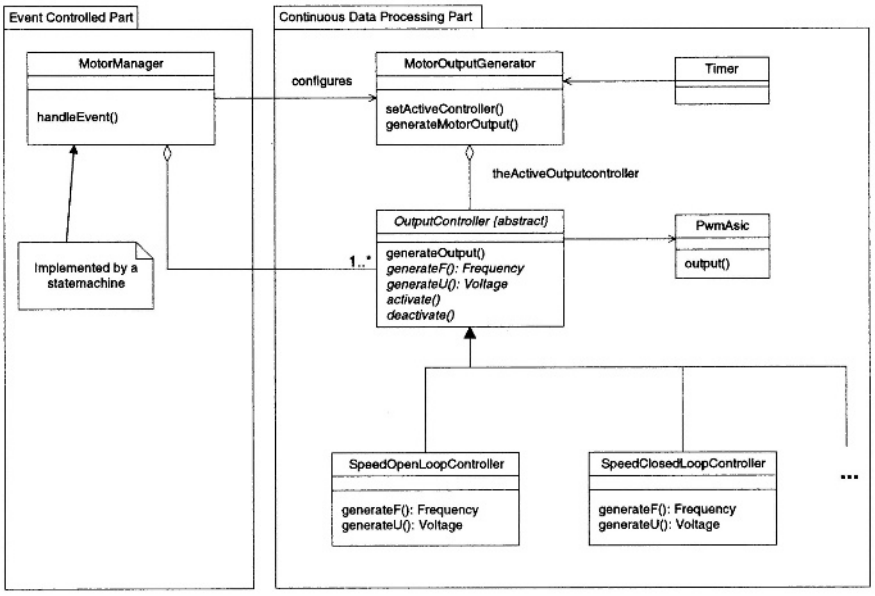


Fig. 5. Package diagram for central parts of the model

The method `generateMotorOutput` in the class `MotorOutputGenerator` will be activated periodically by a timer interrupt every 2 ms. This function will delegate the generation of motor output data to the `generateOutput` operation of the current active output controller object.

```
MotorOutputGenerator::generateMotorOutput()
{
        // ….
        theActiveOutputController->generateOutput();
        // ….
}
```

The `generateOutput` operation calculates the control parameters and sends them to the ASIC controlling the motor.

```
OutputController::generateOutput()
{
          frequency= generateF();// pure virtual
function
          voltage= generateU(); // pure virtual function
          thePwmAsic->output(frequency,voltage);
}
```

The class `MotorManager` receives the events relevant to motor control and determines which of the outputcontroller-objects has to be the current outputcontroller. This class implements a finite state machine. Whenever the `MotorManager` wants to set another outputcontroller active, it calls the `setActiveController` method in `MotorOutputGenerator` with the new `OutputController` object as parameter.

The internals of the method `MotorOutputGenerator::setActiveController` exposes the chosen solution to fulfil the "bumpless transfer" requirement. The solution was to enhance the standard Strategy pattern with functionality to enable this bumpless transfer between the operation modes by implementing two additional operations `activate` and `deactivate` in the `OutputController` class.

```
MotorOutputGenerator::setActiveController
             (OutputController: newController)
{
          controllerInfo
             = theActiveOutputController->deactivate();
          theActiveOutputController= newController;
          theActiveOutputController-
>activate(controllerInfo);
}
```

The current controller is deactivated and the returned information is used to initialise the new controller object, which starts with the same conditions as the previous controller.

Benefits

It is our experience, that the two-part model compared to our former approach is beneficial with respect to several of the design evaluation criteria mentioned in [3], namely "separation of concerns and locality", "perspicuity of design" and "abstraction power".

In our former software the event handling and output computation were intermixed. One of the drawbacks was that it was difficult to understand and test the individual output control algorithms.

In the new architecture the focus on "the system's operational modes and the conditions that cause transitions from one state to another"[3] has been localized in the event-controlled part. Separated from this and localized in the continuous processing part is the focus on the behaviour of the individual output control

algorithms. This also leads to a higher degree of reuse, since the output control algorithms tend to be unchanged in different products utilising the same inverter control hardware.

When looking at a specific functionality, the clear cut between event-handling and continuous behaviour has made it easy to determine in which part of the model, the functionality should be placed.

The two-part model has given raise to a vocabulary (`OutputController`, `Setpoint`, `MotorManager`, etc), which makes it easier for the developers to understand and make additions to our architecture.

References

1. *The Centre for Object Technology (COT)* is a three-year project concerned with research, application and implementation of object technology in Danish companies. The project is financially supported by The Danish National Centre for IT-Research (CIT) and the Danish Ministry of Industry. (`www.cit.dk/COT`)

2. Mary Shaw and David Garlan: Software Architecture: Perspective of an Emerging Discipline. Prentice-Hall, 1996.

3. Mary Shaw: Comparing Architectural Design Styles. IEEE Software, November 1995.

4. Finn Overgaard Hansen, Hans Peter Jepsen: Designing Event-Controlled Continuous Processing Systems. Presented at Embedded Systems Conference, Europe 1999. A revised version to appear in the April 2000 issue of Embedded Systems Programming, Europe.

A Product Line Architecture for a Network Product

Dewayne E. Perry

Electrical and Computer Engineering
The University of Texas at Austin
Austin TX 78712 USA
+1.512.471.2050
perry@ece.utexas.edu
www.ece.utexas.edu/~perry/

ABSTRACT

Given a set of related (and existing) network products, the goal of this architectural exercise was to define a generic architecture that was sufficient to encompass existing and future products in such a way as to satisfy the following two requirements: 1) represent the range of products from single board, centralized systems to multiple board, distributed systems; and 2) support dynamic reconfigurability.

We first describe the basic system abstractions and the typical organization for these kinds of projects. We then describe our generic architecture and show how these two requirements have been met. Our approach using late binding, reflection, indirection and location transparency combines the two requirements neatly into an interdependent solution – though they could be easily separated into independent ones.

We then address the ubiquitous problem of how to deal with multiple dimensions of organization. In many types of systems there are several competing ways in which the system might be organized. We show how architectural styles can be an effective mechanism for dealing with such issues as initialization and exception handling in a uniform way across the system components.

Finally, we summarize the lessons learned from this experience.

0.1 Keywords

Software Architecture Case Study, Dynamic Reconfiguration, Distribution-Free Architecture, Architecture Styles, Multiple Dimensions of Organization

1 Introduction

This study represents a snapshot in the process of constructing a generic architecture for a product line of network communications equipment. The intent of the project was to create the software architecture for the next generation of

F. van der Linden (Ed.): IW-SAPF-3, LNCS 1951, pp. 39–52, 2000.

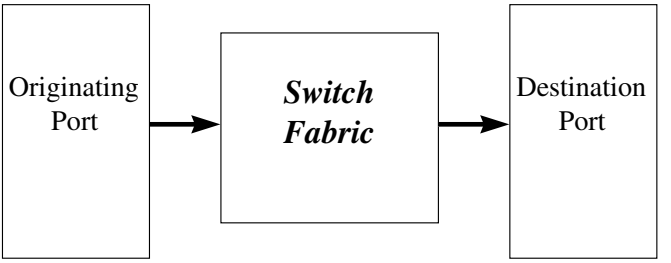


Fig. 1. Basic Abstraction: Connection. A connection consists of an originating port connected via a switch fabric to a destination port.

products in this domain using the existing set of products as the basis for that effort.

The project began in an unusual way: the software architecture team came to research looking for help with their project. They had the domain expertise for the network product as well as experience as architects and system builders. I had experience as a software designer and architect in a variety of domains (but not this one) as well as research expertise in software architecture in general and product line architectures in particular. The result was a fruitful collaboration that lasted about 9 months.

Several caveats are in order before we proceed to discuss the issues and their solutions.

- First, we do not describe the complete architecture. Instead, we concentrate only on the critical issues relevant to the product line and the implications of these issues.
- Second, we present only enough of the domain specific architecture to provide an appropriate context for the part of the architecture and the issues we focus on.
- Third, we address only three architectural issues and describe several architectural techniques that solve these issues in interesting ways.
- Fourth, we do not here discuss issues of analysis such as performance. The architects already did that very well and, as a researcher, that was not where my expertise was applicable (it was in the areas of basic abstractions and generic descriptions). The primary performance issue related to the discussion below was about the efficiency of current commercially ORBs — the one selected appeared to satisfy the required constraints.
- Fifth and finally, we do not provide a full evaluation of the architecture (for example, how well did it work in the end) primarily because, for a variety of reasons, the project was not completed. We do, however, offer the positive consensus of the project architects and their satisfaction with the resulting solutions we discuss below.

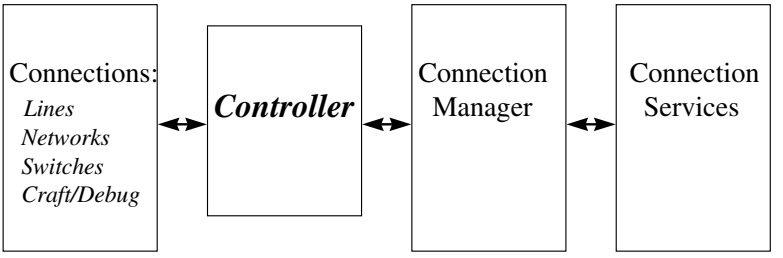


Fig. 2. Basic Hardware/Software System: consists of four logical elements: connections, controllers, connection manager and connection services.

We first provide the context for the study (the product line domain, the current and desired states of the product line, and a basic view of the products). We then explore the implications of the selected system goals and what is needed at the architectural level to satisfy these goals. On this basis, we the describe our architectural solutions and the motivation behind our choices. Finally, we summarize what we have done and lessons we learned in the process.

2 Product Domain

The product line consists of network communication products that are hardware event-driven, real-time embedded systems. They have high reliability and integrity constraints and as such must be fault-tolerant and fault-recoverable. Since they must operate in a variety of physical environments, they are *hardened* as well.

These products are located somewhere between the house and the network switch. They may sit on the side of a building or on some other outside location (for example, a telephone pole), or partly there and partly near a network switch, depending on how complicated the product is (that is, depending on the complexity of the services provided and the number of network lines handled).

The current state of the products in this product line is that each one is built to a customer's specifications. Evolution of these products consists of building both the hardware and software for new configurations.

Central to a satisfactory architecture are the fundamental domain abstractions. They provide the basic organizing principles. Here the key abstraction is that of a *connection*. A connection consists of an originating communications line port connected through a switch fabric (appropriate for the type of network service provided) to a destination port. The connections range from static ones (which once made remain in existence until the lines or devices attached to the ports are removed) to dynamic ones (which range from simple to very complex connections that vary in the duration of their existence) — see Figure 1.

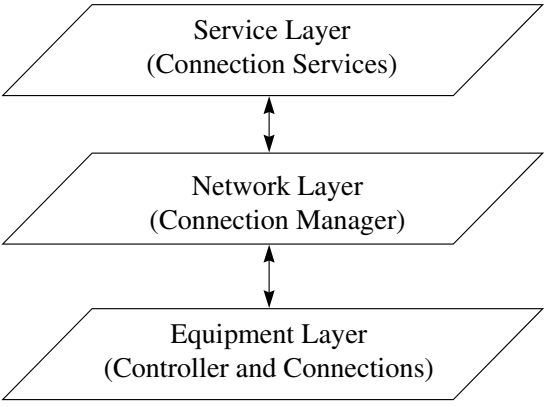


Fig. 3. Typical Domain-Specific Architecture: a structure of three layers consistent with the standard network model.

The typical system structure for these products (see Figure 2) consists of a set of connections such as communication lines, switches, other network connections, and craft and debugging interfaces. These devices have various appropriate controllers that are handled by a connection manager which establishes and removes connections according to hardware control events and coordinates the services required to support the connections.

Figure 3 shows a typical architecture for such network communication products layered into service, network and equipment layers. Within each layer are the appropriate components for the functionality relevant to that layer.

3 Basic System Goals

The basic requirements for the product line architecture we seek are:

- Requirement 1. To cover the large set of diverse product instances that currently exist and that may be desired in the future
- Requirement 2. To support dynamic reconfiguration so that the products existing in the field can evolve as demands change for new and different kinds of communication.

Thus the desired state of the product line is that products can be reconfigured as needed with as little disruption as possible (but not requiring continuous service). For the hardware, this entails common interfaces for the various communication devices and plug compatible components. This part of the project was addressed by the hardware designers and architects. For the software, this entails a generic architecture for the complete set of products and software support for dynamic reconfiguration of the system. This part is what we addressed.

The first question then is how do we create a generic architecture that covers the entire range of products in the product line — that is, how do we satisfy requirement 1? These products range from simple connection systems that consist

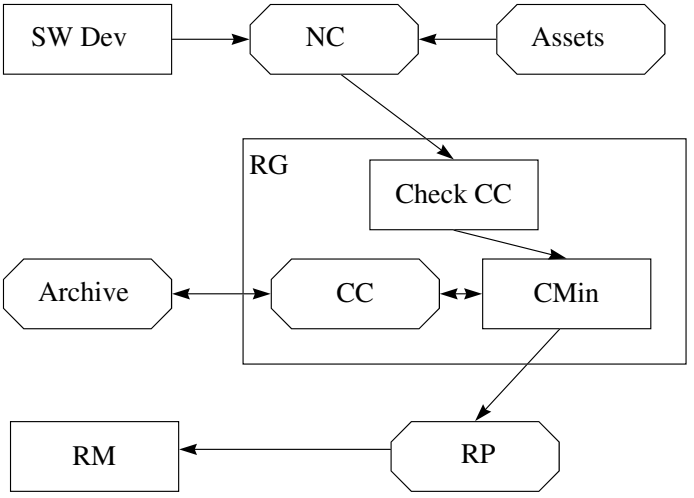


Fig. 4. Reconfiguration: Reconfiguration Generation is shown in detail: new architectural configuration (NC), check consistency and completeness (Check CC), minimize configuration (CMin), reconfiguration package (RP), and current configuration (CC) ; Reconfiguration Management (RM) is shown in detail in figure 5.

of a processor, associated controllers and devices, to complex connection systems that consist of multiple processors, associated controllers and devices which may be distributed over several locations.

The main question is how do we handle this range of variability in component placement and interaction? If we address the issue of distribution at the architectural level, then that implies that distribution is a characteristic of all the instances. What then do we do with simpler systems? A separate architecture for each different class of system defeats the goal of a single generic architecture for the entire product line.

One answer to this problem of variability is to create a *distribution independent* architecture [4] (requirement 1.1) and thus bury the handling of the distribution issues down into the design and implementation layers of the system. In this way, the distribution of components is not an architectural issue.

However, this decision does have significant implications at the architectural level about how the issues of distribution are to be solved. First, the system needs a model of itself that can be used by the appropriate components that must deal with issues of distribution. For example, the component handling system commands and requests must know where the components are located in order to schedule and invoke them. Thus, second, we need a command broker that provides *location transparent* communication, that is configurable, that is priority based and that is small and fast. So not only do we get a view of the architecture where distribution is not an issue, we get a component view of

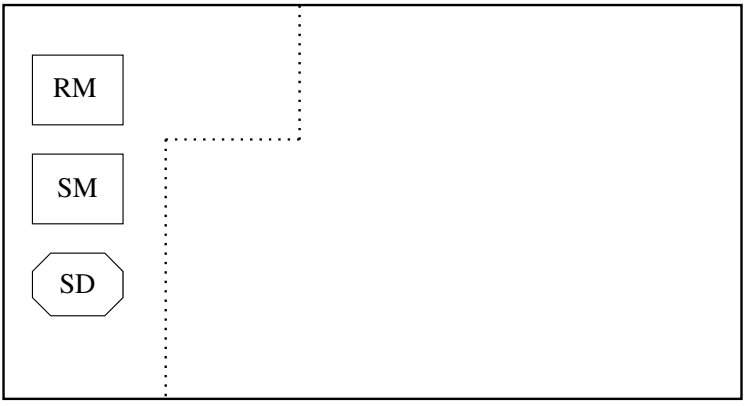


Fig. 5. Reconfiguration Components: System Model (SM), System Data (SD), and Reconfiguration Manager (RM). The dotted line separates the domain specific part from the reconfiguration and distribution-independence parts of the architecture.

communication where distribution is not an issue either. Finally, the components need to be location independent in order to be useful across the entire range of products.

To satisfy requirement 2 for dynamic reconfiguration, it is necessary only to minimize down time. We do not need to provide continuous service. However, we need to be able to reconfigure the system *in situ* in any number of ways from merely replacing line cards to adding significantly to the size and complexity of a system (for example, changing a simple system into a complex distributed one) in the hardware and from changing connection types to adding and deleting services in the software.

As with the issue of distribution, reconfigurability requires a model of the system and its resources, and obviously, a reconfiguration manager that directs the entire reconfiguration process both systematically and reliably. For this to work properly, the components have to have certain properties akin to location independence for a distribution-free system. In this case, we need configurable components. We shall see below that these necessary properties can be concisely described in an architectural style [1].

4 Architectural Organization

By and large, a product line architecture is the result of pulling together various existing systems into a coherent set of products. It is essentially a legacy endeavor: begin with existing systems and generalize into a product line. There are of course exceptions, but in this case the products preceded the product line.

The appropriate place to start considering the generic architecture is to look at what had been done before. In this case we drew on the experience of two

teams for two different products and use their experience to guide us in our decisions.

As in many complex systems, there are multiple ways of organizing [2] both the functionality and the various ways of supporting nonfunctional properties. In this case, we see two more or less orthogonal dimensions of organization: system objects and system functionality. System objects reflect the basic hardware orientation of these systems: packs, slots, protection groups, cables, lines, switches, etc. System functionalities reflect the things that the system does: configuration, connection, fault handling, protection, synchronization, initialization, recovery, etc.

Given the two dimensions, the strategy in the two developments was to organize along one dimension and distribute the other throughout that dimension's components. In the one case, they chose the system object dimension, in the other they chose the system functionality dimension. In the former, the system functionality is distributed across the system objects — for example, each system object takes care of its own initialization, fault tolerance, etc. In the latter, the handling of the various system objects is distributed throughout the system functions — for example, initialization handles the initialization for all the objects in the system.

Both groups felt their solutions were unsatisfactory and were going to choose the other dimension on their next development.

Our strategy then was to take a hybrid approach: choose the components that are considered to be central at the architectural level and then distribute the others throughout those components — a mix and match approach. The question then is how to gain consistency for the secondary components that get distributed over the architectural components. We illustrate the use of architectural styles as a solution to this problem in two interesting cases below.

5 Architectural Solution

We discuss our solutions to the issues we have raised and show how these different solutions fit together to resolve these issues in interesting ways. We discuss first the architectural components needed to support dynamic reconfigurability. We then discuss how distribution independence can be integrated with reconfigurability. We then delineate the general shape of the domain-specific part of the generic architecture and describe how the entire architecture fits together. We then discuss the two primary connectors: one for reconfiguration and one for system execution. Finally, we present two architectural styles to illustrate the distribution of the secondary dimension objects across the primary dimension of organization.

5.1 Reconfiguration (Requirement 2)

Reconfiguration is split into two parts: reconfiguration generation and reconfiguration management. The reconfiguration generator is outside the architecture

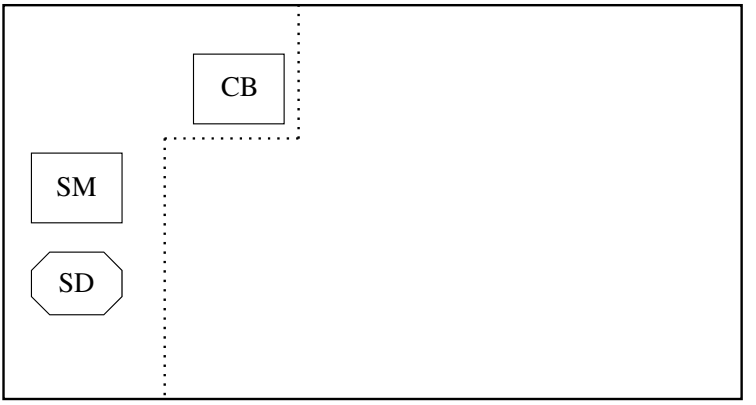


Fig. 6. Distribution Independence Components: System Model (SM), System Data (SD), and Command Broker (CB).

of the system and ensures two primary requirements: first (requirement 2.1), that the reconfiguration constraints for completeness and consistency of a configuration are satisfied; second (requirement 2.2), that the configured system is minimal [3], a requirement due to both space and time limitations.

The question arises then as to where this part of reconfiguration should be. Given the space and economic considerations of the systems, we chose to have the consistency checking and reconfiguration minimization done outside the bounds of the system architecture.

In Figure 4, a new architectural configuration (NC) is created by combining new components from software development (if there are any) with existing assets and passing them to Reconfiguration Generation (RG). The new configuration is then checked for consistency and completeness (Check CC). Once it is established that those constraints are satisfied, the new configuration is compared against the current configuration to to determine which architectural components need to be added and deleted (C Min). The result is a Reconfiguration Package (RP) which is passed to the Reconfiguration Manager (RM) containing the instructions for dynamically reconfiguring the software part of the system.

To satisfy requirement 2 for system reconfigurability, we have the three components illustrated in Figure 5: the reconfiguration manager (RM), the system model (SM) and the system provisioning data (SD).

The system model and system data provide a logical model of the system, the logical to physical mapping of the various elements in the system configuration, and priority and timing constraints that have to be met in the scheduling and execution of system functions.

The reconfiguration manager directs the termination of components to be removed or replaced, performs the component deletion, addition or replacement, does the appropriate registration and mapping in the system model, and handles

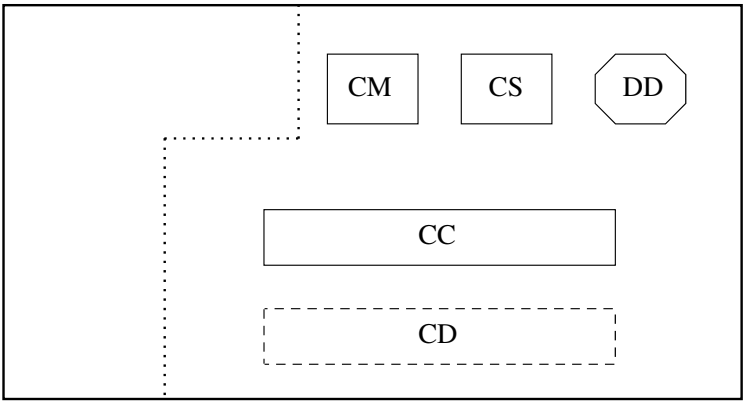


Fig. 7. Domain-Specific Components: Connection Manager (CM), Connection Services (CS), Dynamic Data (DD), Connection Controller (CC), and Connection Devices (CD).

startup and reinitialization of new and existing components. Special care has to be taken in the construction of the reconfiguration manager so that it can properly manage self-replacement, just as special care has to be taken in any major restructuring of the hardware and software.

5.2 Distribution Independence (Requirement 1.1)

For the satisfaction of the distribution independence requirements, we have the three components illustrated in Figure 6: the command broker (CB), the system model (SM) and the system provisioning data (SD). Note that the system model and the system provisioning data are the same as in the reconfiguration solution.

The command broker uses the system model and system provisioning data to drive its operation scheduling and invocation. System commands are made in terms of logical entities and the logical to physical mapping is what determines where the appropriate component is and how to schedule it and communicate with it.

5.3 The Domain Specific Components

For the domain-specific part of the architecture we have chosen as the basic architectural elements the connection manager (CM), the integrity manager (IM), the connection services component (CS), the dynamic data component (DD), the connection controllers (CC), and the connection devices (CD). These components represent our choices for the architectural abstractions of both the critical objects and the critical functionality necessary for our product line. Of these, the integrity manager is a logical component whose functionality is distributed throughout the other components shown in Figure 7.

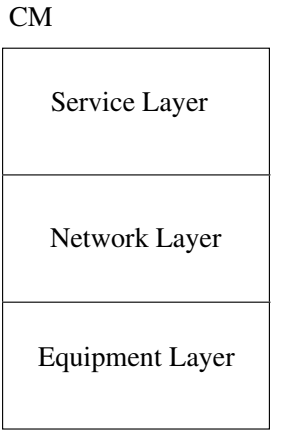


Fig. 8. Domain Specific Component Decomposition. The traditional layering forms the basis of the subarchitectures of several of the basic domain specific components. Here we see the decomposition of the Connection Manager (CM).

While we have not used the typical network model as the primary organizing principle for the architecture, it does come into play in defining the hierarchy or decomposition of several of the basic domain specific system components: the connection manager (Figure 8 illustrates this decomposition of this component), the connection services, and the connection controller.

5.4 Connectors

The reconfiguration interactions shown in Figure 9 illustrate how the reconfiguration manager is intimately tied to both the system model and the system provisioning data. This part of the reconfiguration has to be handled with care in the right order to result in a consistent system. Further, the reconfiguration manager interacts with itself and the entire configuration as well as the individual components of the system: terminate first, preserve data, reconfigure the system model and system provisioning, and then reconfigure the components. There are integrity constraints on all of these interactions and connections.

A logical software bus provides the primary connector amongst the system components for both control and data access. The manager of the bus is the command broker. There are other connectors as well, but they have not been necessary for the exposition of the critical aspects of the generic architecture. There are both performance and reliability constraints that must be met by this primary connector. How to achieve these constraints was well within the practicing architects expertize and as such is not, as performance issues in general are not, within the scope of our research contributions nor the scope of this paper.

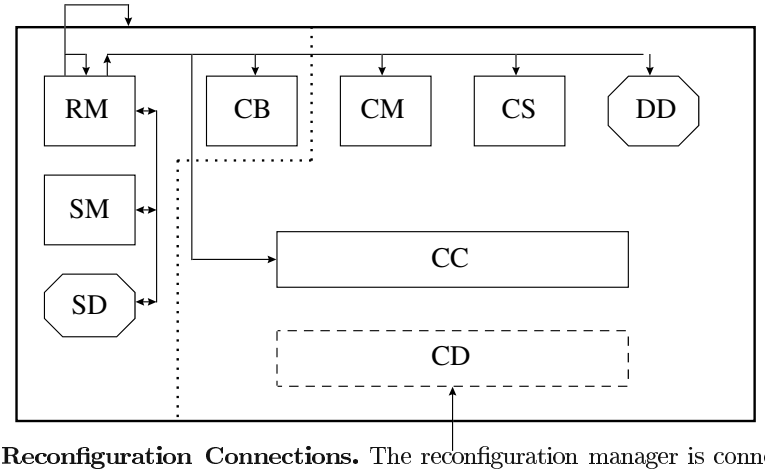


Fig. 9. Reconfiguration Connections. The reconfiguration manager is connected in various ways to all the components in the system, including itself and the system as a whole.

5.5 Architectural Styles

So far we have delineated the primary architectural components derived from the goals for reconfiguration or distribution independence, or from the two domain-specific dimensions of organization possible for this product. For those domain-specific components not chosen, we provide architectural styles to ensure their unform implementation across all the chosen components. We present two such styles as examples: a reconfigurable component style and an integrity management style.

The reconfigurable component architectural style that must be adhered to by all the reconfigurable components has the following constraints:

- The component must be location independent
- Initialization must provide facilities for start and restart, rebuilding dynamic data, allocating resources, and initializing the component
- Finalization must provide facilities for preserving dynamic data, releasing resources, and terminating the component

We had also mentioned earlier that the integrity manager was a logical component that was distributed across all the architectural components. As such there is an integrity connector that hooks all the integrity management components together in handling exceptions and recovering from faults. We had also indicated that the part of the integrity management would be defined as an architectural style that all the system components had to adhere to. This style is defined as follows:

- Recover when possible, otherwise reconfigure around the fault
- Isolate a fault without impacting other components
- Avoid false dispatches

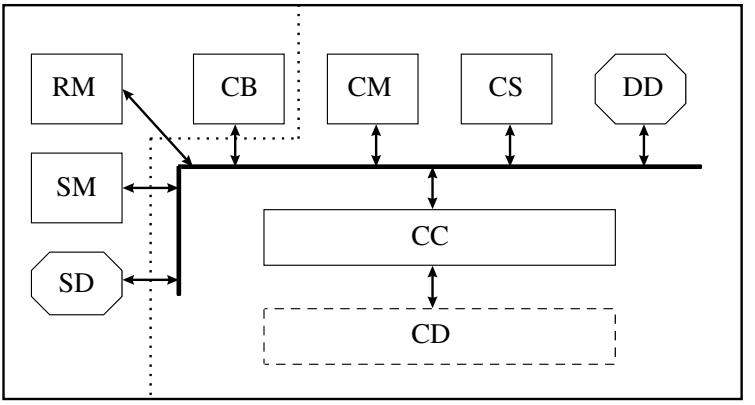


Fig. 10. Architectural Connections. A software bus provides the primary control and data connectors among the system components.

- Provide mechanisms for inhibiting any action
- Do not leave working components unavailable
- Enable working in the presence of faults
- Recover from single faults
- Protect agains rolling recoveries
- Collect and log appropriate information
- Map exceptions to faults
- Enable sequencing of recovery actions

Styles such as these function as requirements on architectural components to guarantee a consistent and uniform set of component properties and behaviors.

6 Summary and Lessons

We have explored several interesting techniques to achieve a generic architecture that satisfied both the domain-specific requirements and the product-line architecture requirements.

To delineate the appropriate domain-specific components, we used a hybrid approach in which we selected what we considered to be the critical elements from two orthogonal dimensions of organization. We then defined architectural styles to ensure the consistency of the secondary components distributed throughout the primary components.

We defined a logical software bus, subject to both performance and reliability constraints, as a general connector among the components. These constraints are especially important where the underlying implementation and organization is distributed across several independent physical components.

To achieve the appropriate goals of the generic architecture covering a wide variability of hardware architectures and enabling dynamic reconfiguration, we

chose a data-driven, late binding, location transparent and reflective approach. This enabled us to solve both the problem of centralized and distributed systems and the problem of reconfiguration with a set of shared and interdependent components.

As to lessons learned:

- There are many ways to organize an architecture, even a domain specific one. Because there are multiple possible dimensions of organization, some orthogonal, some interdependent, experience is a critical factor in the selection of critical architectural elements, even when considering only functional, much less when considering non-functional, properties.
- It is important for any architecture, design or implementation to have appropriate and relevant abstractions to help in the organizing of a system. An example in this study is that of a connection as the central abstraction. Concentration on the concepts and abstractions from the problem domain rather than the solution domain is critical to achieve these key abstractions.
- Properties such as distribution-independence or platform-independence are extremely useful in creating a generic product line architecture. They do, however, come at a cost in terms of requiring architectural components to implement the necessary properties of location transparency or platform transparency.
- Architectural styles are an extremely useful mechanism in ensuring uniform properties across architectural elements, especially for such considerations as initialization, exception handling and fault recovery where local knowledge is critical and isolated by various kinds of logical and physical boundaries. These styles define the requirements that the system components must satisfy to guarantee the properties and behaviors of the secondary components.

Acknowledgements

Nancy Lee was my liaison with the architectural group on this project. She helped in many ways, not the least of which was making project data and documents available for me to write up this case study. The system architects on the project as a whole were very tolerant of an outsider working with them. However, we achieved a good working relationship combining their domain expertize with my research investigations together with a willingness to explore alternative possibilities.

Thanks also to Ric Holt at Waterloo for his comments and suggestions.

References

1. Dewayne E. Perry and Alexander L Wolf. Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes*, 17:4 (October 1992)
2. Dewayne E. Perry. Shared Dependencies. In *Proceedings of the 6th Software Configuration Management Workshop*, Berlin, Germany, March 1996.

3. Dewayne E. Perry. Maintaining Minimal Consistent Configurations. Position paper for the 7th Software Configuration Management Workshop, Boston Massachusetts, May 1997. Patent granted.
4. Dewayne E. Perry. Generic Architecture Descriptions. In *ARES II Product Line Architecture Workshop Proceedings*, Los Palmas, Spain, February 1998.

Railway-Control Product Families: The Alcatel TAS Platform Experience

Julio Mellado, Manuel Sierra, Ana Romera, and Juan C. Dueñas[1]

Alcatel Transport Automation Systems Spain
Depto Ingeniería de Sistemas Telemáticos, Universidad Politécnica de Madrid
{ julio.mellado, manuel.sierra, airomera }@alcatel.es,
jcduenas@dit.upm.es

Abstract. Railway-control systems must cope with strict dependability requirements. Typically, a product or installation is composed by several applications, different operating systems, and a number of hardware pieces. The range of combinations is large and thus the range of products in the family. In order to keep this figure small, handle the development process efficiently (unavoidably delayed by the validation efforts required) and to isolate software and hardware elements, Alcatel Transport Automation Systems (TAS) took a strategic decision: to choose an operating system able to hide hardware variability, and provided by a third party. This paper describes the concept of product family used by Alcatel TAS, organized around an operational platform for applications in the product line, reports the experience carried out in Alcatel Spain, in porting existing railway control applications to the Alcatel TAS Platform (an operating system developed by Alcatel) and discusses some advantages of this approach.

Introduction

Railway-control systems form an application domain characterised by these facts:

- dependability requirements such as safety, reliability, availability and responsiveness are the most important, and are regulated by governmental authorities. Failures could lead to human life losses.
- there is a great diversity of existing systems already working, since these control systems are embedded in a large network system (the railway system), each of its parts evolving and ageing at different speeds. The control applications must cope with a large number of different hardware elements.
- products have a long life span (more than 20 years), so changes and updates must be done incrementally.

[1] The work of Juan C. Dueñas has been partially developed in the the project "Engineering Software Architectures, Processes and Platforms for System-Families" (ESAPS) ITEA 99005/Eureka 2023, and has also been partially funded by Spanish CICYT under the project "Integrated development for distributed embedded systems".

F. van der Linden (Ed.): IW-SAPF-3, LNCS 1951, pp. 53-62, 2000.

Examples of railway-control systems are the following:

- Interlocking Control Systems. These systems control the train movements in a single station. Their main goal is to guarantee that the trains move within the station safely and to avoid the risk of collision. See [6] for an exhaustive description of products from different companies.
- Traffic Control Systems. They survey the railway traffic in a whole line. Their main goal is to schedule the traffic in order to minimise the delays.
- Automatic Train Protection Systems: They control the speed of trains according to the status of the track signals.
- Train Position Detection Systems: Axle counters.

Many of the current products in this market are reaching the end of their life. The key reasons for this fact are their inability to compete with the growing demands of market (in quality/price relationship, performance, commercial criteria), and the decreasing availability on specific hardware components (some of them are partially obsolete or surpassed) [5]. Thus, for example, software-controlled electronic systems are replacing electro-mechanical ones since years, due to their flexibility, lower price and easier maintenance. The operating systems in the control part were usually tightly coupled to the hardware platform, so most of them are proprietary [6].

In this article we describe the general requirements of the railway-control systems domain. Then we describe the TAS Platform and how it satisfies the needs of railway-control applications. Finally, we show how the TAS Platform concept works in practice describing the experience of a real project. The TAS Platform operating system is the basis for all the Alcatel family of railway-control products, regardless the hardware they are operating on top.

Requirements

There are several requirements shared by all the products in the railway-control domain. Some of them originate from customers: since failures in the system can lead to severe damages and loss of human lives, the system itself becomes a safety critical application [2]. So, for example, the system must be able to detect malfunctions and enter a safe state in that case (fail-safe systems). Obviously, time constraints are very strict for these reactions. Another important requirement is reliability: the system must work twenty four hours a day, seven days a week.

The railway-control systems must fulfil specific safety standards. Inside the European Union, the CENELEC [3] Norm Set is the main standard for these systems. This norm classifies the systems according to five “integrity levels”, each one with different safety requirements. Every system working in a real installation needs a customer validation. The possibilities for optimisation of the validation effort have increased due to the “cross-acceptance” principle within the European Union. A railway administration can accept a system (or a part of it such as the hardware or the operating system) validated by another European administration.

Redundancy has been chosen as the main strategy to ensure availability requirements. The configurations of several processors working in parallel and synchronising to compare the values of their inputs, their outputs and some of their internal data are typical in this domain. In "two out of two" configurations

(abbreviated 2oo2), two processors are working together, and the safe state is entered if their results are different. In "two out of three" (abbreviated 2oo3) configurations, if a processor does not agree with the other two, it is disconnected. Redundancy is not only applied to the processor but to the hardware links with the external equipment. The communication channels are typically doubled, although a “One Channel Safe Transmission Protocol” has been accepted in the CENELEC Standard recently. Even software is made redundant: several implementations of algorithms with the same functionality are running on different processors. Choosing the degree of redundancy, and which elements will be duplicated, leads to different products in the product line.

Another set of requirements depend on the environment where the railway-control system will operate: each customer (railway administration) has specific requirements, such as different signalling rules, different external equipment, et cetera. Companies present in more than one country have to deal with these differences, adapting their products to the customer needs or developing in each national unit products targeted to specific markets, which is inefficient in terms of effort and cost.

Maintainability in the long term is also important: usually, railway control systems last for many years. In order to improve the functionality of the systems, hardware updates are necessary. But these updates should have a minimum impact in the application software. The two main reasons to avoid changes to the software are: the application internal complexity and the cost of validating the changes made.

And the last, but not least, are the economical requirements. Profitability of a certain product or product line depends not only on their quality, but on issues like "time to market", and, even more important, on the product portfolio. This is tightly related with the cost reduction of the development by using plugable components for either hardware and software parts. Reusing as much as possible of the existing systems in the new ones, and incorporating commercial and well-proven technology are good approaches to be competitive.

Concept of the TAS Platform

The TAS Platform is an operating system that has gathered the previous work done in Alcatel with the AEOS Operating System. The first goal of the TAS Platform is to serve as an standard basis for the whole Alcatel TAS family of products. Another main goal is to make the Alcatel TAS software applications independent from the hardware. An important point is that the TAS Platform has been developed as the result of the experience with the existing systems; this made visible the need of identification of commonalities and variabilities in the whole family of products, the need to factor out the common part (mainly the TAS Platform), and also the need to reduce its development effort.

The TAS Platform affects the architecture of the systems that use it, and implicitly is defining the architecture of a family of products. Using the terms defined by [8], it can be described as a “variance-free architecture” in which the differences between the products are not relevant from the architectural point of view. The TAS Platform can also be considered as a market-oriented product rather than a customer-oriented product, which is a typical characteristic of a product family, as [4] points out.

The Figure 1 describes the generic architecture of all the systems that use the Platform.

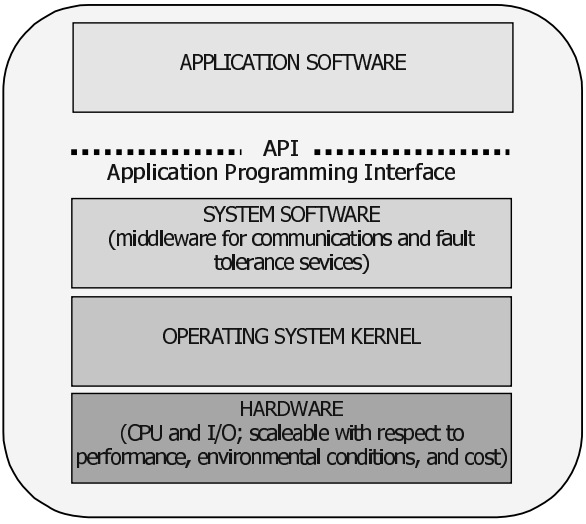


Fig. 1. Basic System Architecture.

It is important to note that in this case, the "family of products" does not hold different variants of the same system, with small changes to their functionality. Rather, the main common characteristic of the products in the family is that all of them belong in the same domain (and therefore share certain requirements, either functional or not) and are able to operate on top of the same platform that satisfies their common requirements. Thus, "family of products" is defined as the set of applications in the domain that can be executed on a certain operational environment. This environment is the Alcatel TAS Platform, which offers to the applications complete safety mechanisms to operate and allows system redundancy (2oo2 and 2oo3) without changing the application software.

Certainly, this definition reveals the strong connection between the concept of product line and the different ranking of requirements for different domains. One reason to define "family of products" in this way is that dependability requirements are much more important to cope with than pure functional ones; thus, validation activities can take more effort than -for example- implementation tasks, so once the main problem in the domain (guarantying dependability) is solved, populating the family (porting more applications to the Platform) involves less effort and thus shorter time to market.

The generic architectural schema that appears in the Figure 1 shows that applications only see a software interface offered by the Platform, ignoring all the underlying details. This API conforms to the standard POSIX [1] interface. The approach, a common basis adaptable to different products, is similar to other industrial experiences such as [7], but with the difference that no assumptions are made about the application software architecture in the Platform. A reason for this is that the TAS Platform is not only for the fully new developments made inside Alcatel TAS, but also for the evolution of already existing systems.

The "system software" box in the Figure 1 represents the Platform services. A more detailed view of the system architecture is presented in Figure 2. For the operating system kernel, a COTS ("commercial-of-the-shelf") has been chosen: the Chorus real-time microkernel, produced by Sun Microsystems. The microkernel approach provides a larger flexibility, since the kernel is composed by a flexible set of interchangeable pieces that can be selected depending of the specific requirements of products. Another COTS have been used to implement the TCP/IP and X.25 protocols.

With this basic architecture, and after domain analysis activities have been performed, several points of variation are identified:

the internal communication mechanisms, encapsulated in the communication system (CS). By means of using message queues, timers and signals, this module makes possible that the application software runs on different processors synchronously (all the processors do the same and have the same internal data). These redundancy and location transparencies have to be configured depending on the application requirements.

the fault tolerance mechanisms, encapsulated in the fault tolerance layer (FT), that implements the safety requirements, collecting all the fault indications. It is responsible for the consistency of the data between the different processors and the recovery mechanisms. It is possible to configure different reactions for fault indications.

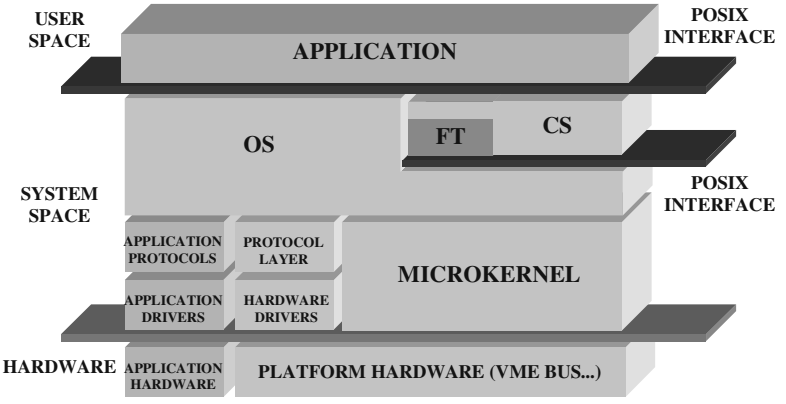


Fig. 2. TAS Platform services structure.

The applications can fulfil their safety and availability requirements either using the full services of the TAS Platform (contained mainly in CS and FT) or using only the most basic ones (those in the microkernel). In the first case, the TAS Platform sees the application as a set of tasks that communicate using message queues, and the application does not care about the processor redundancy and error handling. In the second one, the application has to deal with all the task error handling, the communication between tasks, and the processor redundancy (guaranteeing that the processors running in parallel are synchronous). The first approach means less effort for the applications developers, that can concentrate on their specific problems after assuring that the requirements and the architecture of their applications can be supported by the TAS Platform. For example, in the interlocking systems, route

treatment is a very critical and complex problem at the application level, but has no relation with the synchronisation among the replicated processors at all, which is a typical task for the TAS Platform.

The TAS Platform implements the replica deterministic concept to assure the safety execution in redundancy systems: 2oo2 and 2oo3. Redundancy systems are composed of redundant tasks that contain the application software running in all the processors of the system. The implementation of the concept ensures that all the redundant tasks receive the same data from non-redundant task (those that only run in one processor) or external systems, and that all of them task start from the same points and at the same time. The implementation is based on the synchronisation of redundant tasks of the application and the voting mechanism of the messages received by these tasks.

The voting mechanisms are implemented in the FT layer and are configured by application developers to run in 2oo2 or 2oo3 configurations. The 2oo2 voting mechanism takes care about differences of two external messages and the possibility that one of them could be lost. This case is a fault and the system stops. The 2oo3 voting mechanism expects three identical messages, but the system continues working if two of messages are identical, establishing a majority vote among the three messages expected.

Due to the concept of deterministic replica, the TAS Platform operating system implements its own scheduler for the tasks of the application (redundant or not). The application tasks use CS services to define the preemption points needed by the TAS Platform scheduler and exchange data through message queues. These message queues are POSIX message queues with, again, a voting mechanism.

The TAS Platform operating system provides mechanisms to use the multitasking scheduler of the CHORUS microkernel that is also conform with the replica deterministic concept. The scheduling units are logical groups of application tasks (Taskset elements). The Taskset elements run concurrently, and all the application tasks inside run in a replica deterministic way.

The hardware elements are encapsulated by means of drivers that lay within the system space. In this case, the operational environment that is specified by the customers determines the selection of hardware elements. As new hardware components must be included in the system, new drivers are written.

As a conclusion, the TAS Platform is used in the whole Alcatel TAS family of products due to its flexibility, adaptability, hardware independence, applicability to different elements in the railway-control domain (there are working examples of interlockings, axle counters, and automatic train protection systems), acceptance by different railway customers in different countries (Austria, Spain, Canada, Germany, Finland, etc.), and fulfilment of safety requirements.

Report of Experience. The Porting Project

A train control system for track sections supervision has been developed by Alcatel TAS Spain to control small and medium-size railway stations. The application is structured in several modules (see Figure 3), all of them programmed in ANSI C language:

- MMI: man-machine-interface.
- SIM (Simulator): software system that allows the simulation of field elements.
- OS (Operating System): embedded operating system with a POSIX interface.
- LD (Local Diagnosis): check module (only for development purposes).
- CM (Control Module): includes supervision logic, operator command processing and field data acquisition software.

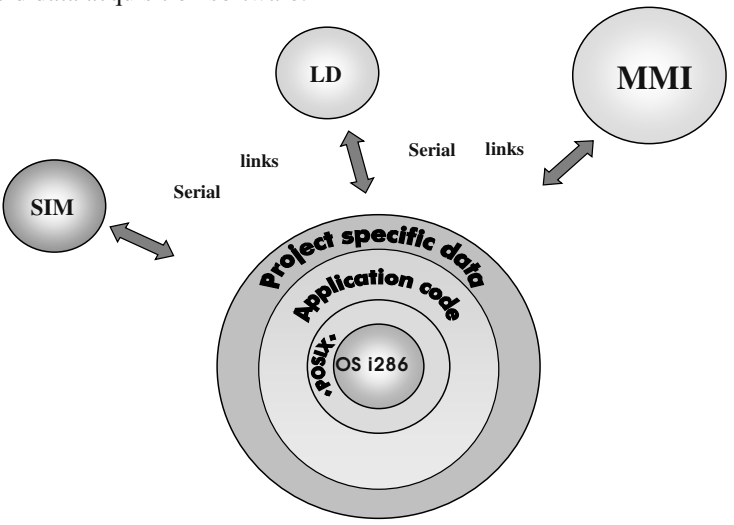


Fig. 3. Application architecture.

The train control system application was chosen to be ported to the Alcatel TAS Platform, in order to enhance its capabilities. This new system would be scaleable to adapt at different line sizes, and modifiable to be easily ported to new hardware platforms.

During the porting process, some changes have been done to integrate the application onto the new operating system and to use Platform services. It is very important to ensure that changes do not modify the current functionality, nor the safety conditions of the product. Therefore all the procedures according CENELEC EN 50126, 50129 and 50128 have to be considered.

The relationships between the application ported and TAS Platform are shown in the Figure 5. The application uses the TAS Platform services (synchronisation and voting) to assure a replica deterministic execution. Furthermore, the application increases its performance with the concurrent execution of some tasks (interface tasks with main application).

For the porting process the following steps have been performed:

1. Training a team of application software developers on the new TAS Platform system, the new development environment and the special programming characteristics for use of TAS Platform services. For this purpose, the experts of the TAS Platform make design documentation available in side Alcatel TAS Spain. It is important to note that the creation and dissemination of this documentation within the company helps in the organisation of the internal "know-how", and to

codify the best practices. For example, the documentation includes an implicit decision table for the design of the redundancy mechanisms.
2. Elaboration by the Spanish team of a system requirements specification of the train control system to the TAS Platform.

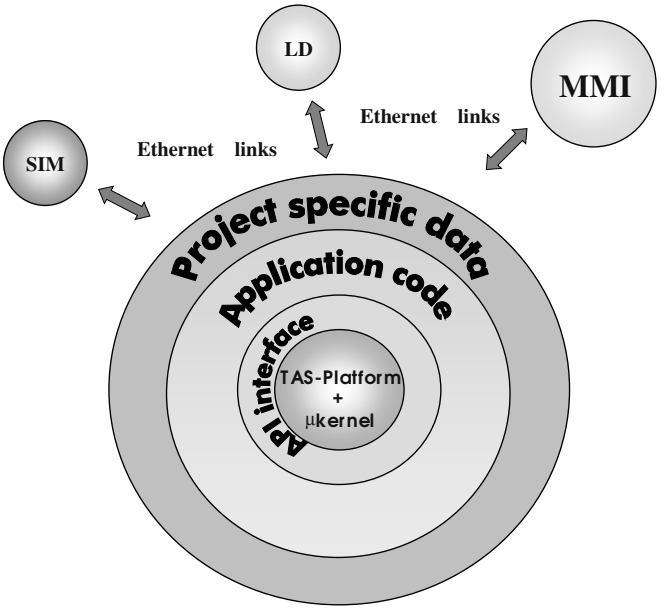


Fig. 4. Application system based on TAS Platform.

3. Development of the first ported version, taking the main part of the already available code and making the required changes. The programming language is in both cases ANSI C. The operating system interface is POSIX on both cases, and has not changed significantly. Additional modules have been written to manage Ethernet-based communications instead of serial protocols. A new TAS Platform Configuration, specific for the train control Application, has been created.
4. Testing of this first prototype with a set of well-defined test protocols used in the train control Application system to validate the system functionality. Tests were performed with simulators, fed with real system input data. The results obtained were compared with the former one. The results of the tests were successful in all cases.

Conclusions and Future Work

The train control system project has reached its objectives of delivering a new system based on Alcatel TAS Platform. Besides, the project achieved these goals:
– Reuse of the existing software, maintaining all functionality.
– Increase reliability and scalability of the product.

– Application software independent of hardware.
– Shorter time and lower cost development of the new product family.

After the development of the first version of the train control system on TAS Platform, the future work is addressed to improve the functionality and the performance of the new system, considering new hardware configurations.

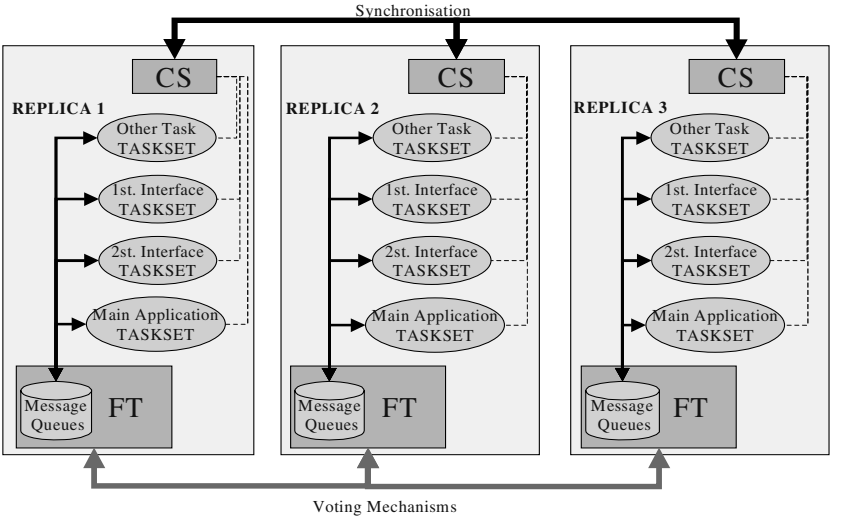


Fig. 5. Relationship between the Ported Application and the TAS Platform.

Acknowledgements

Ana Romera, Julio Mellado, Manuel Sierra wish to thank their employer, Alcatel TAS Spain, for giving them the possibility of take part in this interesting and encouraging project. The valuable support of TAS Platform Team, and the tremendous expertise of his members has been a constant during all the project, making it possible to happen. We thank all of them.

References

[1] ANSII/IEEE Std 1003.1 and ISO/IEC 9945-1, Information Technology - Portable Operating System Interface (POSIX ®) - Part1: System Application Interface (API) [C Language], 2nd Edition, 1996

[2] Bowen, J., Isaksen, U. y Nissanke, N., (1997) System and Software Safety in Critical Systems, Technical Report RUCS/97/TR/062/A, Reading University, Computer Science Department England.

[3] Comité Europeén de Normalisation Électrotechnique CENELEC (1997). prEN 50128, Railway applications - Software for railway control and protection systems.
[4] Dolan, T. Weterings, R., Wortmann, J. C. (1998). Stakeholders in Software-System Family Architectures. Development and Evolution of Software architectures for Product families. Second International ESPRIT ARES Workshop, Las Palmas de Gran Canaria, Spain, 1998, Proceedings. Springer Lecture Notes in Computer Science 1429.
[5] Doppelbauer J., Lennartz K., Veider A., Warlitz J.,(1998). "Basisystem für signaltechnisch sichere Anwendungen", Signal+Draht 10/98, pp 8-12
[6] Maschek U (1997) Elektronische Stellwerke - ein internationaler Überblick, Signal+Draht 03/97, pp 8-23
[7] Meekel, J. , Horton, T. B., Mellone, C. (1998). Architecting for Domain Variability. Development and Evolution of Software architectures for Product families. Second International ESPRIT ARES Workshop, Las Palmas de Gran Canaria, Spain, 1998, Proceedings. Springer Lecture Notes in Computer Science 1429.
[8] Perry, D. E. (1998) Generic Architecture Descriptions for Product Lines. Development and Evolution of Software architectures for Product families. Second International ESPRIT ARES Workshop, Las Palmas de Gran Canaria, Spain, 1998, Proceedings. Springer Lecture Notes in Computer Science 1429.

Discussion Report "Business" Session

Günter Böckle

Siemens ZT SE
Otto Hahn Ring 6
D-81739 München, Germany
Guenter.W.Boeckle@mchp.siemens.de

The “Business” session of the workshop covers the basic question to be answered when a product line approach is adapted by an organisation:

- Is it worth the effort? Will the product line approach really deliver the expected benefits?
- What will the assets for reuse be so that these benefits can be achieved?

Two presentations showed two complementary methods to answer these questions. The first shows a fast, simple method to determine the potential of a product-line approach, which helps to achieve a fast decision, whether to apply this approach for a certain set of products. The second shows a detailed method, which determines the scope of a product line, and the benefit, which can be achieved by adopting the product-line approach. A tool is presented which helps in determining the scope of the product line.

The presentations lead to a lively discussion, which concentrated, on two major topics:

- The tools and methods presented
- Why are we doing this kind of work – what is the difference between SW and other products?

The domain potential analysis uses the standard analysis methods, however some different ways to calculate the results so that the potential of a product line approach with respect to the risk involved can be visualized. Similarly, PuLSE-BEAT uses standard economic analysis methods, extended and enhanced for product lines. There are some more methods around, but those are too coarse-grained to deliver sufficiently reliable results. For both methods the reliability of the results depends of course on the reliability of the input used for their application.

This started a discussion about the fact that many organizations just do not have the data, which are necessary for evaluating the potential of product lines and for scoping them. However, this must not prevent this activity – one should start and use guestimates where necessary! These are systematic and disciplined methods and they will show what is missing and provide the way for structured analysis. Of course they cannot replace measurement programs in organizations for providing the data, but they can show what data is needed. Perhaps some people may see their creativity impaired – but the application of such methods will get people organized so that they can use their creativity in a planned and structured way. And after all: having no data which can be used for deciding about a product line approach shows clearly that there is something wrong in the organization and that there is a need to improve!

F. van der Linden (Ed.): IW-SAPF-3, LNCS 1951, pp. 63-64, 2000.

The models for investment to be applied for product line engineering cannot be restricted to either linear or exponential models – both may be necessary, depending on the particular product line, the health of the organization, and the political conditions. Both, high-level analyses and detailed analyses are needed, but the results are error-prone and have to be taken with a grain of salt. A factor of three should be taken into account. However, the organization will learn and apply better measurement methods – there are many examples of successful product line engineering. And experience shows that even if there is a factor of two between prediction and result, others will get attentive and will copy the methods and new product lines will appear. But due to the vague input data and the fact that such analyses are not well-known this still appears to many as kind of a black science.

This directed the discussion to the statement that we are not working in our own field – we are specialists in information science and engineering but not economists or business administrators! Why shouldn't we leave this task to the experts? Well, these experts do not understand the complexity of software and the difference from non-software products; so we have to get engaged and address all topics, including business and organizational issues. We have to interact with the rest of the world and convince them. We can learn from the automobile industry where platform technology is used to reduce time to market significantly. However, the design space in the mechanical industry is more controllable, there is no direct translation to software, e.g. because of the different ways to implement software. Another difference is in the value of software; you cannot determine the value of software as easily as that of mechanical parts. The value of a piece of software can change from one moment to the next, there is no base value for financial calculations. It is not possible to sell used software like used mechanical items, it just has no worth any more. In mechanical systems the major cost is in production while in software it is in specification.

But there are advantages in such differences; software can be upgraded in the field. An example is that we can make an MP3 upgrade by just downloading new software. Very profound changes of the market will occur, this is a great opportunity for us. Even individual products may become product lines. Car manufacturers think about such methods and events; what we need is a good business model for software.

PuLSE-BEAT — A Decision Support Tool for Scoping Product Lines[1]

Klaus Schmid†, Michael Schank*

†: Fraunhofer Institute for Experimental Software Engineering (IESE),
Sauerwiesen 6, D-67661 Kaiserslautern, Germany
Klaus.Schmid@iese.fhg.de

*: University of Kaiserslautern, Computer Science Department
Michael@schank.de

Abstract. Determining the scope of a product line is a core activity in product line development, as it has a major impact on the economic results of the project. Only recently an approach was proposed that allows to base this decision explicitly on a detailed analysis of the relevant business goals [2]. As the approach requires gathering and evaluating a lot of data, tool support becomes mandatory. In this paper, we describe the tool PuLSE-BEAT, which was specifically developed to address this need.

1 Introduction

1.1 Motivation

Recently, software product lines have been accepted as a key approach for large scale software reuse. While huge benefits are often associated with this approach (cf. [1]), reaping them requires thorough planning and identification of the best opportunities. This happens during the scoping phase of a product line project. *Scoping* consists of identifying the most promising sub-domains relevant to the product line and, early on, identifying those assets, which, when developed for reuse, provide the highest benefit.

The PuLSE™-method (Product Line Software Engineering[2], cf. [4]), which is an approach to product line engineering developed at the Fraunhofer IESE, contains an explicit scoping step, called PuLSE-Eco. In the following section, we will briefly describe the approach used by PuLSE-Eco for performing scoping. A more detailed description can be found in [2]. The specific approach taken by PuLSE-Eco for determining the scope relies heavily on the thorough analysis of data describing the benefits associated with making certain characteristics of the product line reusable. While, initially, we performed this analysis by hand (i.e., using Excel™), we now developed a tool for assisting this analysis. This tool is called PuLSE Basic Eco Assistance Tool (PuLSE-BEAT). In this paper, we will concentrate on discussing the

[1] This work has been partially funded by the ESAPS project (Eureka Σ! 2023 Programme, ITEA project 99005).

[2] PuLSE is a registered trademark of the Fraunhofer IESE

F. van der Linden (Ed.): IW-SAPF-3, LNCS 1951, pp. 65–75, 2000.

requirements for a successful scoping support tool and how PuLSE-BEAT satisfies these requirements.

1.2 Related Work

Here, we will briefly describe other published scoping approaches. However, as these approaches – to our knowledge – do not possess any specialized tool support (i.e., tool support is on the level of non-specialized graphics and table management software), we cannot directly compare the approaches on the tool level. Thus, we will restrict ourselves to a comparison of the approaches:

Most work on the scoping issue, so far, has been done in domain engineering. In particular, methods such as Synthesis [9] or Organizational Domain Modeling (ODM) [10] introduced scoping approaches, but these remained on a rather abstract level. Perhaps the most stringent approach for domain scoping is provided by the domain scoping framework [7], but this approach fails to link the derivation of the scope explicitly with the business goals. A major difference with PuLSE-Eco is also that this approach is strongly based on the notion of a domain while ours is based on the notion of precise products. This, we believe, is a major distinction that needs to be made in order to truly address the business goals as they are usually centered around products. The only scoping work we know of that is also based on the concept of a line of products is the work by Withey [11]. However, this approach is restricted to the business goals of maximizing the return on investment (ROI).

Besides the software-oriented approaches, also approaches from other disciplines exist. Especially interesting here is the work by Robertson and Ulrich [8]. However, while [8] is closest to our work, it still centers around a pre-defined set of criteria, while our approach is open to a wide range of business goals.

Work on relating technical characteristics to external requirements is also relevant in this context. Quality Function Deployment (QFD) is probably the most well-known approach in this area [6]. Also the notion of a product map as we describe it here shows some commonalities with the matrices used in QFD.

This paper is structured as follows: in the following section we introduce the underlying scoping approach: PuLSE-Eco. Then in Section 3.1 we will give a description of our tool and discuss how it lives up to the requirements that are set by the method and its application context. In Section 3.2, we will then discuss some additional features of the tool that simplify its application in the process. Section 4, then briefly discusses an example for using the tool and Section 5 concludes.

2 The Scoping Method: PuLSE-Eco

The PuLSE-Eco approach is described in detail in [2]. This approach aims at identifying early on those characteristics that shall be directly supported by the product line architecture (respectively the reuse infrastructure) in order to maximize the benefit of the product line approach. As different organizations have different goals in following a product line approach, the ability to flexibly incorporate these goals was a key issue in the definition of the method. As high level business goals are too abstract to use them directly for the decision making process they are refined into *benefit* and *characterization functions* using a GQM-like approach [3] in the step *develop evaluation functions*. The functions are derived in such a way that the benefit functions can be expressed in terms of the characterization functions. Benefit functions capture a certain kind of benefit (e.g., effort reduction) that can be gained by having a certain characteristic within the scope of product line development (i.e., having its functionality in a reusable form). Characterization functions in turn describe a certain attribute of a product/characteristic pair (e.g., effort of developing a certain characteristic in the context of a specific product). These characterization functions are then used to explicitly define the benefit functions [5]. This derivation of benefit functions and characterization functions from a business goal is captured in a *derivation table*. These functions can be roughly equated with metrics in the GQM-approach, where the two are related such that the characterization functions correspond to directly (subjectively) measurable properties, while benefit functions

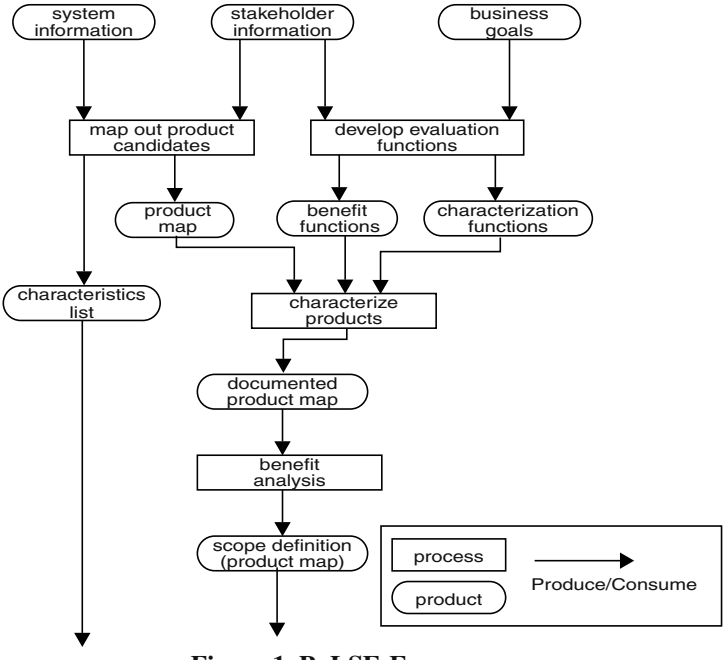


Figure 1. PuLSE-Eco process

correspond to complex metrics the value of which can be computed using the characterization functions.

The main visual device for organizing this information is the *product map*. This is a matrix with the various characteristics that are relevant to products in the product line on the left axis and the various products and the characterization functions on the top (cf. Figure 2). When this table has been completely filled out, a numerical analysis can be performed on the gathered data in order to identify an optimal scope definition.

Figure 1 gives a short overview of the PuLSE-Eco process. The first step is to identify the characteristics that are relevant to the products in the product line. A structured list is elicited from the stakeholders, existing systems, and the product plan. Similarly a list of the systems potentially relevant to the product line must be developed. These may be existing, future, or even hypothetical applications. Note, that in order to perform this analysis we are explicitly looking at individual systems in the product line and not at market segments or the like. The elicitation of this information is performed in a structured manner in the step *map out product candidates*. As a result also a list of the major functions relevant to the products in the product line is developed.

Once this has been done, the main axes of the product map have been established. Next, it is generally appropriate to identify which characteristics will be needed in which product. For this usually a special characterization function called `req` is used. Now, based on the business goals of the product line project, benefit functions are derived, which operationalize the goals and describe the benefit of having a certain characteristic inside the scope. The benefit functions in turn are described using characterization functions. This is performed in the manner described above in the step *develop evaluation functions*. After the operationalization of the goals has been performed, the values for all characterization functions need to be gathered from the stakeholders. This data elicitation is performed in the step *characterize products*. This step is completely determined by the products and characteristics elicited previously and by the characterization functions defined before. One way of enacting this step is to simply give the tool to the stakeholders and ask them to enter this information. Then, using the definition of the benefit functions, their values are computed and this is then used as basis for deciding which characteristics to develop for reuse and which ones not (benefit analysis). This step of deciding what to develop for reuse and what not is called *benefit analysis*. It can be performed semi-automatically. All computations for deriving the benefits of the characteristics are performed by the tool and those characteristics where product lines can be expected to be beneficial relative to a threshold are marked automatically. The threshold values need to be determined based on the benefit functions. A typical way is to define the benefit functions in such a way that they range from 0 to 1 and the break-even is at 0.5 (i.e., for a characteristic that is evaluated to 0.5 there is no difference in developing it in a reusable or in a non-reusable manner).

Now, PuLSE-BEAT supports three different scopes with successively higher thresholds. The typical approach is to have these different scopes as possible candidates, likely candidates, and strongely recommended candidates with thresholds of 0.5, 0.6 (to add a risk premium), 0.75 (in this case a strong advantage for product

line development can be expected). However, as benefit functions are chosen differently for each application and thresholds are relative to the selected benefit functions, the thresholds may be needed to be adopted on a case by case basis. The tool itself makes no assumptions about the threshold values, nor about their interpretation. Its relevant at this point to observe that the benefit functions are defined such they are monotonouos in the values of the characterization functions. Further, they are not particularly sensitive to a single parameter. Thus with a spacing of the thresholds like the one given above, a change of scope will not occur based on a single data error except for true borderline cases. However, as technical relations may exist among the individual characteristics which may go beyond the gathered information, the proposed scope may need to be revisited by an expert anyway.

While it is not central to PuLSE-Eco, the same principles, that are applied for determining the scope can also be used for selecting those applications that are best suited for the product line. This form of product portfolio management is also supported by PuLSE-BEAT.

3 The Scoping Tool: PuLSE-BEAT

3.1 Core Capabilities

Based on the description of PuLSE-Eco given above some major requirements for tool support can be derived:

- Support of the various workproducts and their interrelations
- Support all steps of the scoping process
- Easy to use for both domain experts and method experts
- Can be provided to customers at low cost

The approach relies on several different workproducts. Most important among them is the product map, but also derivation tables for the benefit and characterization functions are needed and detailed descriptions of the defined functions need to be recorded. All these workproducts are heavily interrelated and these relationships need to be managed.

Additionally, three different phases can be distinguished in the method: developing the information, i.e., identifying characteristics and deriving characterization functions (done by domain experts and product line experts); gathering the data (performed by domain experts), and scoping (performed by product line experts). The tool is to be applied in all three phases. Thus, it is necessary that the system runs on an easy to access platform.

Based on these core requirements for the tool the decision was made to develop it based on Excel™ using Visual Basic for Applications. Given the widespread availability of Excel, we believe this platform will be available to all potential users. Additionally, given the high reliance of the PuLSE-Eco approach on tabular notations (product map, derivation tables, etc.) this allows to use directly the various Excel table handling facilities, e.g., for printing and online-formatting, without any extra-cost.

Given this environment, the decision was made to map the various workproducts to worksheets, i.e., each one is implemented as a separate table. However, as all worksheets are contained in one work-book, all information relating to a single scoping project is always directly available and can, for example, be loaded with a single command. Overall the documentation for a scoping project consists of the following worksheets (actually the tool uses additional sheets for internal purposes, however, their description is outside the scope of this paper):

- Product Map (external representation)
- Benefit functions: this table accumulates all the definitions for benefit functions
- Auxiliary functions: this table contains all the definitions for auxiliary functions (Auxiliary functions can be sometimes introduced to map values among different value ranges.)
- Characterization functions: this table aggregates all characterization functions
- Derivation table: for each refined goal one derivation table is generated

The tool provides consistency management and traceability between all these worksheets. This is actually a difficult task as the same benefit or characterization function may occur in the operationalization of several goals, but obviously they shall appear in the product map only once, as data for them shall also be gathered only once. This is also the underlying reason for having explicit worksheets that aggregate the definitions of the various functions. These sheets are also used to keep track of how often these functions are referenced. Thus, e.g., a characterization function is actually removed from the product map only when the last reference to it is removed from a derivation table.

Note, the distinction that is made between the external and internal representation of the product map. The internal representation is never directly manipulated by hand, thus it is not listed above. The reason for this distinction is to allow the tool users to manipulate the data in a more intuitive form, that is, characterization functions can actually be defined using arbitrary alphanumerical values, which are only internally mapped onto numerical values for computation. This allows for example to describe whether a certain feature is required in a specific product using 'X' and '_'. While this seems to be a simple feature it requires some effort to implement and it increases readability of the product map considerably (cf. Figure 2). Additionally, this approach simplifies the checking of input data.

Based on the provided data the tool can compute the benefit functions and based on thresholds decide whether a certain characteristic belongs inside or outside of the scope. This approach can also be used with the tool for deciding whether a certain product should be considered part of the product line or not by refining product-related goals into benefit functions and computing the benefit of having a certain product in the product line.

3.2 Additional Features of PuLSE-BEAT

In the preceding section, we outlined the core requirements on a scoping tool and how PuLSE-BEAT satisfies them. In this section, we will discuss some additional features of the tool. While we think they are not vital to the tool's usability, they substantially simplify its usage in real world projects:

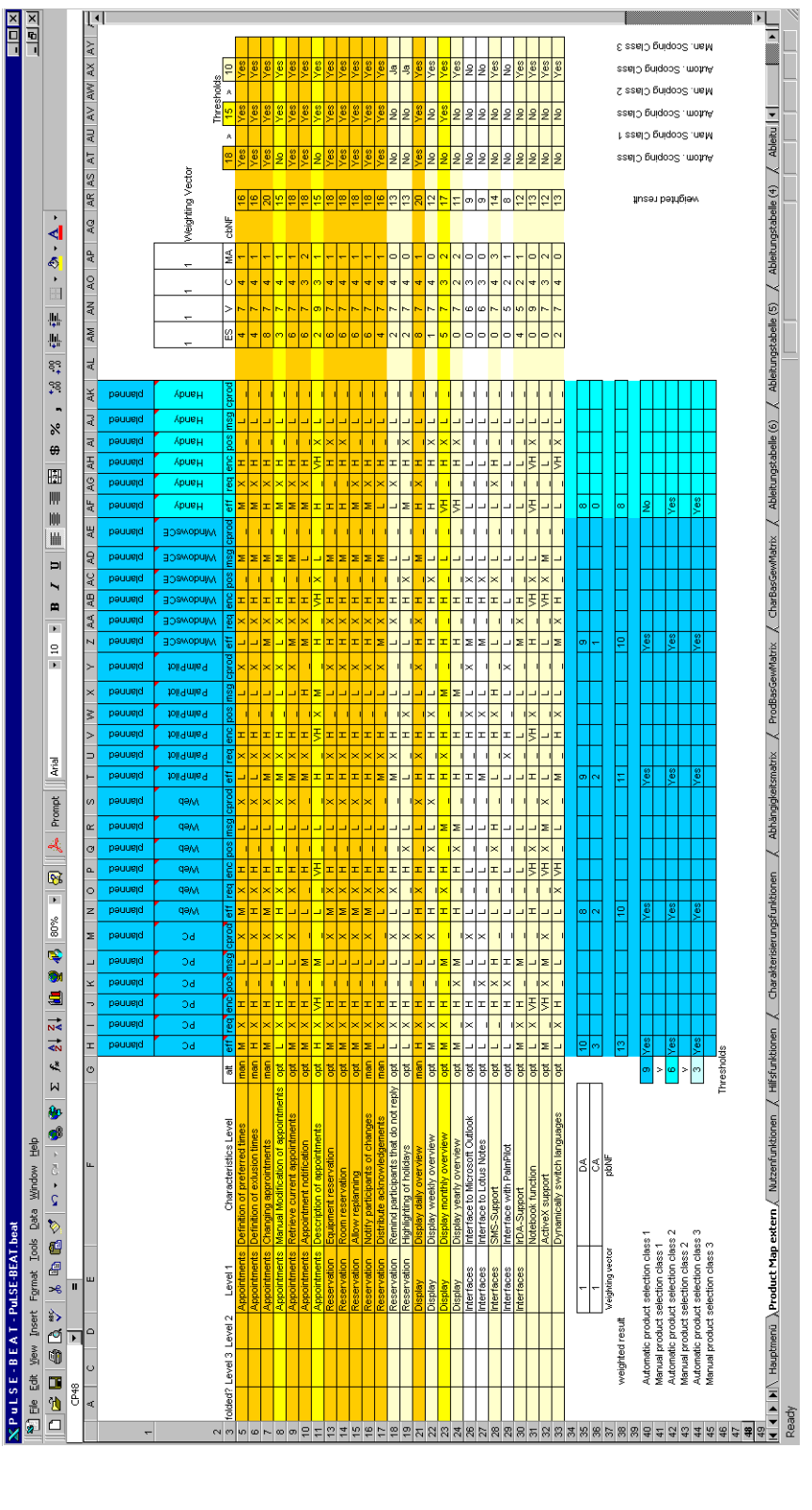


Figure 2. Screenshot of a scoped product map

An important aspect in real-world scoping projects is the need to deal with a large amount of information, thus a support tool needs to be able to simplify the data handling. In particular, in the case of PuLSE-Eco a large number of characteristics needs to be handled for industrial-size systems. The tool supports this with functions that allow to cluster the characteristics hierarchically into groups and to collapse these groups again into single higher level characteristics, if required. With this feature a better overview of the product line characteristics can be gained. How values for individual characteristics are combined into values for complete groups can be explicitly defined on a case by case basis.

The tool allows to work with several scopes simultaneously. The semantics of these scopes are not predefined. One possible usages is: different scopes for: performing domain analysis, inclusion in the product line architecture, and implementation of reusable assets. Another one: scopes for upper- and lower-bounds for inclusion in the reuse infrastructure.

Another important feature is the ability to generate reports, which can be used as a basis for external review. Providing this feature was particularly simplified by the choice of Excel/VBA as implementation platform as this allows to export the report information directly into Word, where it can be either directly printed or used as a basis for further work.

4 Using PuLSE-BEAT

While in the preceding sections, we discussed the main features of the tool, in this section we briefly illustrate how the various features synergetically work together to smoothly support derivation of the scope.

The first step is to gather the information about the relevant characteristics and products as described in Section 2. Besides the short description shown in Figure 2, also long descriptions can be entered. They are added as comments to the cells (thus they appear when moving the mouse-pointer over the cell) and can be also shown in reports. Descriptions can be attached to both characteristics and products. Additionally, characteristics can be later on reordered to arrive at a logical ordering and grouping.

In the example given in Figure 2 a hypothetical product line of time planning software is shown. This product line example is based on the hypothesis of a client-server system (only the clients are part of the product line). The client software should run homogeneously on different platforms (plain PC, as Web-application, on Windows CE systems, etc.). The different platforms impose some restrictions such that the relevant features will vary from product to product. The characteristics that were identified were grouped into categories: core time-planning, facility reservation, display, and interfaces.

At this point the report-feature of the tool can be used. It generates automatically a report of the information captured in the tool so far. The reports generated in this way are a preferred basis for reviews by the stakeholders. This type of reviews are iterated until the product and characteristic lists are sufficiently stable.

The next step is to elicit the business goals and to derive from them benefit and characterization functions. As described in Section 2, the approach used to perform this derivation is based on the GQM-approach. The specific approach is described in more detail in [5]. Per goal, which is relevant to the scope selection, one derivation is created and captured in a derivation table (cf. Figure 3). The table captures the goal that triggered the derivation, the type of the goal (goal type), i.e., whether the goal relates to the selection of charactistics for the scope or to the selection of products as part of the product line. Further, the questions are captured that are used to refine the goal. Finally, the benefit functions are captured that are used to operationalize the goal. The characterization functions describe the atomic parts of the benefit functions (cf. Section 2). Additionally, auxiliary functions are used to simplify the description of a benefit function. The derivation is supported by the tool by capturing the relevant information and thus documenting the process and by managing interrelations with other work-products (e.g., by adding columns to the product map that are needed to gather values for the characterization functions).

As stated before, the evaluation approach supported by PuLSE-BEAT can also be used to analyze whether specific products are truely adequate as part of the product line. During capturing the evaluation criteria this is described by identifying the goal "product-based".

In filling out the derivation table, the support for handling of interrelations among the various work-products provided by the tool is the most visible: when adding the benefit, auxiliary, or characterization functions, the user is diverted directly to the appropriate work sheet, where he can fully describe these functions, if they have not yet been defined. When the user is finished with this task, these functions are appropriately entered in the product map. If, for example, a characterization function is defined (and it is not defined as relating only to the characteristic in general), then for each product a column is added for holding the values for this function.

Figure 3 shows the derivation table for determining the impact on effort of making a certain characteristic reusable. In the example, one benefit function (ES) is used, which in turn uses two characterization functions (eff, req). The characterization function *eff* captures the effort needed for creating a single characteristic in the context of a specific

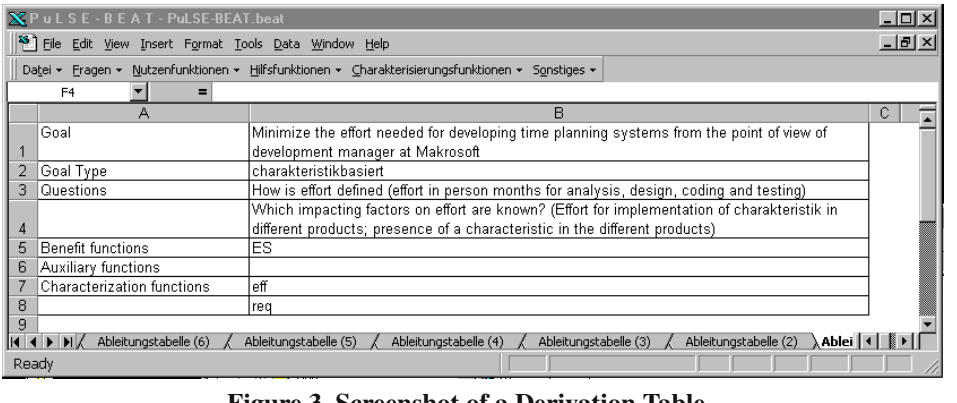


Figure 3. Screenshot of a Derivation Table

product. *req* describes whether a certain characteristic is required for a specific product.

After the product map has been set up like this, the tool can be handed over to the stakeholders to insert the needed data. As they only need to manipulate the external view of the product map, it is possible to hide all other worksheets, so as to prevent accidental manipulation of the data entered so far. At this stage the tool is particularly useful, as it empowers the people to do work off-line (if the meaning of the characterization functions is well understood). The tool ensures that entered values are in the required range and based on the entered data a proposed scope can be automatically computed. This allows the users to easily assess the ramifications of adding or removing specific features to/from products.

After the values have been provided by the stakeholders, all data has been gathered. Now, scoping of the characteristics that should go into product line development can be performed automatically by the tool based on the functions that have been provided. Similarly, an evaluation of the product portfolio can now be performed automatically if appropriate product-based evaluation functions have been defined. Figure 2 shows a product map where scoping has been performed both with respect to the characteristics as well as with respect to the products. Those characteristics that are greyed in Figure 2 are recommended for inclusion in the scope (the darker, the stronger is the recommendation). All products, except the right-most one are in the highest level of adequacy for the product line, only the last one is in the second-highest level, which is an indicator that the inclusion of this product in the product line should be reconsidered.

5 Conclusions

In this paper, we described a tool called PuLSE-BEAT, which was designed to support the PuLSE-Eco scoping approach [2]. In particular, we discussed the main requirements that are posed by the scoping task for a support tool and described how PuLSE-BEAT meets each of them. In addition, we described how the individual features synergetically work together by giving an example of the tools usage.

To our knowledge, this is the first scoping support tool that has been described in literature. We regard as key advantages of our tool its ability to capture the complete information on a scoping effort and to maintain complete traceability among the individual information items. Further, it provides automatic support for performing the scoping decision, thus it also enables the analysis of what-if scenarios in a straightforward manner.

While we believe, that there is still some room for improvement of the tool, we think that at this point all core requirements are addressed. Thus, we plan to use it from now on in our industrial scoping projects.

At this point enhancements to the tool concentrate mainly on improving the usability (ease-of-use, robustness, and speed) of the tool, as the current version covers completely the PuLSE-Eco method, as it was described in [2]. In this context, we see

only room for minor functional extensions, e.g., extending the expressiveness of auxiliary functions.

However, as we do currently work strongly on enhancing and extending the method itself in the context of industrial cooperations, we expect to co-evolve the tool in accordance with future enhancements of the PuLSE-Eco method.

Bibliography

1. L. Brownsword and P. Clements. *A Case Study in Successful Product Line Development*. Carnegie Mellon Software Engineering Institute, CMU/SEI-96-TR-016, 1996
2. J.-M. DeBaud and K. Schmid. *A systematic approach to derive the scope of software product lines*. In Proceedings of the 21st International Conference on Software Engineering, 1999.
3. L. Briand, C. Differding, and D. Rombach. *Practical guidelines for measurement-based process improvement*. Software Process Improvement and Practice Journal, 2(3), 1997.
4. J. Bayer, O. Flege, P. Knauber, R. Laqua, D. Muthig, K. Schmid, T. Widen, and J.-M. DeBaud. *PuLSE: A methodology to develop software product lines*. In Symposium on Software Reusability '99 (SSR'99), May 1999.
5. K. Schmid. *An Economic Perspective on Product Line Software Development*. First Workshop on Economics-Driven Software Engineering Research, Los Angeles, May, 1999.
6. L. Cohen. *Quality Function Deployment*. Addison Wesley, 1995.
7. Department of Defense — Software Reuse Initiative, Version 3.1. *Domain Scoping Framework, Volume 2: Technical Description*, 1995.
8. D. Robertson and K. Ulrich. Planning for product platforms. *Sloan Management Review*, 39(4):19–31, 1998.
9. Software Productivity Consortium Services Corporation, Technical Report SPC-92019-CMC. *Reuse-Driven Software Processes Guidebook, Version 02.00.03*, November 1993.
10. Software Technology for Adaptable, Reliable Systems (STARS), Technical Report STARS-VC-A025/001/00. *Organization Domain Modeling (ODM) Guidebook, Version 2.0*, June 1996.
11. J. Withey. Investment analysis of software assets for product lines. Technical report CMU/SEI-96-TR-010, Software Engineering Institute, Carnegie Mellon University, 1996.

Domain Potential Analysis: Calling the Attention on Business Issues of Product-Lines*

Sergio Bandinelli and Goiuri Sagardui Mendieta

European Software Institute
Parque Tecnológico 204, Zamudio, Bizkaia, E-48170, Spain
Sergio.Bandinelli@esi.es Goiuria.Sagarduy@esi.es

Motivation

Product-lines represent a natural step in the evolution of software development into an industrial practice. A product-line approach intrinsically leads to systematic reuse and reuse is supposed to have a positive impact in business terms: saving development and maintenance costs, time to market reduction, quality improvement, more predictable project execution, etc.

In an industrial context, the decision of adopting a product-line approach must take into account a wide range of factors. Technology is, of course, one of these factors, but it is not necessarily the most important one and, for sure, it is not the "driving factor". The drivers for introducing a product-line approach are generally related to the general company strategy, taking into account market considerations. The product-line technology should be evaluated and used in this business context.

However, this previous analysis is not always performed and there is a tendency to jump directly into the technical implementation of a product-line: architecture, components, middleware technology etc. It is first necessary to reason with discipline on what domain (or sub-domain) is the most appropriate one and on whether the selected domain has the potential to justify the effort.

Not all domains are equally appropriate to be approached as a product line. A successful adoption of product-line approach requires some conditions such as potential demand for similar products, in-house knowledge and experience, existing regulations and standards, etc. A domain potential analysis evaluates the degree to which these conditions exist to serve as a reference for:

- Defining a product-line adoption strategy, and setting realistic goals for it.
- Deciding on the most appropriate domains or sub-domains for a product-line approach.
- Reaching consensus on a shared vision for the domain.
- Evaluating progress in product-line adoption.

Some models are available in the literature to perform this kind of analysis. Most of them are economic models and base the analysis on economic figures (cost vs.

* This work is partially funded by the European Commission under ESPRIT project P28651 PRAISE.

F. van der Linden (Ed.): IW-SAPF-3, LNCS 1951, pp. 76-81, 2000.

savings) to determine the benefits at different levels of reuse granularity: single component, project or whole domain. Other models include some analysis of the level of preparation of the organisation. (See [Lim 98] and [Poulin 97] for a survey of all these models).

The domain potential analysis presented here takes these models as a basis for the analysis of reuse benefits and combines this with a risk analysis. The two combined dimensions, benefits and risks, give an overall picture of the potential of a domain. The combined picture provides a clear indication on whether it is convenient to approach a domain as a product-line in absolute terms and by comparing different domains. In addition, the analysis may be adapted to be used with the available data in the organisation.

A Simple Analysis of the Product-Line Potential in a Domain

The product-line potential in a domain provides an indication of the opportunities that derive form adopting a product-line approach to develop applications in a domain and the ability of the organisation to exploit these opportunities to obtain benefits from them.

The concept of domain that we use is a very broad one. It includes

- the technical description of the domain in terms of the existing and potential applications that share some common features (technology, functionality, etc.),
- the market of the domain (customers, competitors, regulations, etc.)
- the organisational structures that participate in the business.

When identifying a domain all these elements must be taken into account, since all of them take part in the analysis.

The potential analysis is similar to taking an investment decision. This is why both benefits and risks must be taken into account:

1. The benefits are the ones that the organisation expects from the product-line approach.
2. The risks are the ones associated with the introduction of product-line practices in the organisation.

The combined analysis benefits vs. risk gives the complete picture to take an investment decision.

Analysing the Benefits

The right context to analyse the benefits is the set of declared goals of the organisation to embark in a product-line. Depending on these goals, the organisation can give more or less weight to one potential benefit over the others.

The list of these benefits can be very long, including the following ones:

- Higher productivity
- Higher quality
- Higher Reliability
- Faster time to market

- BETTER bid estimation
- Better life-cycle estimates
- More on-time delivery
- Cost improvements and savings
- Improved maintenance

This long list can be generally shortened to the classical better, faster, cheaper:

- Quality improvement
- Time to market reduction
- Cost reduction

Ultimately, all the benefits should (in the short or in the long term) result in economical benefits for the organisation. Some of the benefits, such as reduction of costs, can be directly translated into economic results. Other benefits, such as quality improvement and time to market reduction, have an indirect impact on economic results. As a simplification we consider that quality improvement and reduced time to market have an impact as a reduction of maintenance costs and as an increment on the number of requests that may be satisfied in the same period (more production capacity translated into more units sold).

With these hypothesis, there are three main elements that take part in an economic benefits analysis:

- **INVESTMENTS (I)**: resulting from activities to establish and developing the product-line infrastructure;
- **EXPENSES (E)**: resulting from the activities for maintaining the product-line infrastructure during its life;
- **SAVINGS (S)**: savings achieved as a result of the development of applications using the domain assets, comparing the cost to develop with reuse against the cost to develop without reuse.

The depth of the analysis in terms of investments, expenses and savings can vary according to the data available in the organisation. In general, the process model can guide in a breakdown of the activities so that the effort associated with each of then can be evaluated separately and then aggregated.

The economical analysis may be as simple as considering RoI (Return on Investment) in terms of savings against investments and expenses or may also involve additional analysis regarding when investments and expenses are done and when savings are obtained. In this way, we calculate time sensitive indicators such as the NPV (Net Present Value) or (PI) Profitability index.

Analysing the Risks

The objective of the risk analysis is to understand and quantify the major sources of risk when introducing product-line practices in a domain. The analysis is supported by a risk model. This model identifies a set of risk attributes organised into four risk factors (see Figure 1):

- **ORGANISATION**, with attention to the adequacy of the organisational structures to adopt reuse
- **PERSONNEL**, with attention to staff experience and preparation

- **PROCESS**, looking at the presence of supporting processes for reuse
- **PRODUCTS**, looking at the existence of beneficial reuse characteristics in domain products

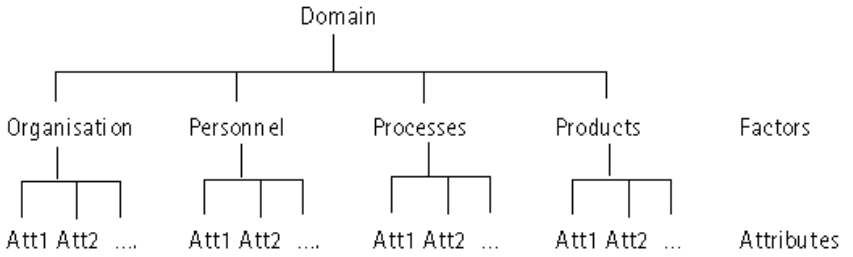


Fig. 1. Structure of the Risk Analysis Model

The risk analysis is performed by rating each of the risk attributes and by giving an impact weight for each attribute [Boehm 89]. The model provides a set of guidelines to consistently interpret each of the attributes. The results from the risk analysis are a risk profile and an aggregated risk level that represents the overall risk for the domain.

Tying It All Together

The potential analysis is completed with the determination of the organisation's attitude toward risk. This is determined through a questionnaire in which the respondent must choose between a set of possibilities regarding investments in a given situation. Three main attitudes are identified [Pike 96]:

- *Risk Taker:* This kind of organisations gives preference to obtaining more benefits at the expense of taking much higher risks.
- *Risk indifferent:* A risk indifferent organisation is ready to take some more risk only if there is a proportional increase in the benefits that may be obtained.
- *Risk averse:* For a risk adverse organisation, the main objective is to reduce risks. The organisation does not look for increments in benefits if this implies taking more risk. A positive result is adequate.

  Most organisations fall in the risk adverse category.

The benefits analysis, the risk analysis and the attitude towards risk are combined in a single graph that summarises the situation for one or more domains. The graph looks like the one depicted in Figure 2, in which the x-axis represents the risk level and the y-axis represents the economic return.

The risk attitude is represented by a line that divides the area into two parts. The part above the line represents the area for which the organisation considers that the domain potential is sufficiently high for an investment on a product-line approach. Under that line, the risk is considered to be too high for the expected benefits and therefore the domain should be rejected as a candidate for product-line investment.

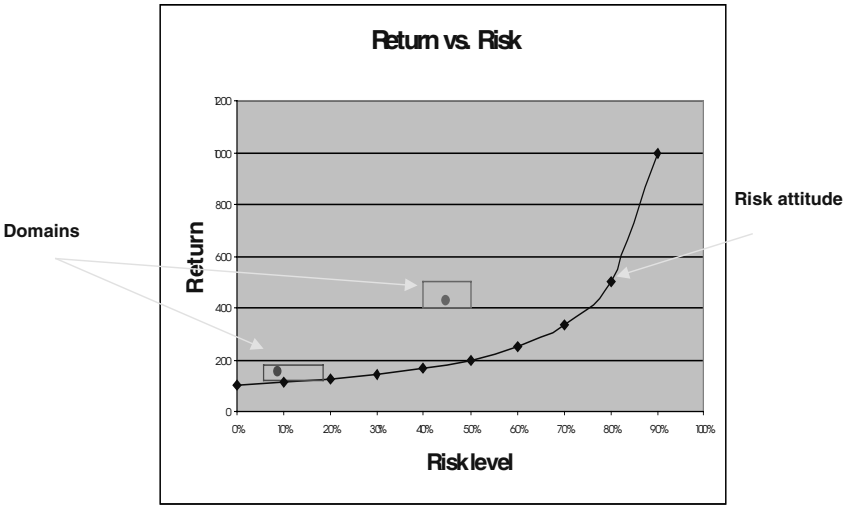


Fig. 2. Return vs. risks

Conclusion

The paper describes a simple, concise and effective way of determining the potential of a domain for introducing a product-line approach. The main contribution of this analysis model is that it summarises much information in one single picture:

1. Support for an investment decision based on benefits vs risk dimensions.
2. Comparative analysis of several candidate domains and sub-domains to help the organisation to concentrate the initial efforts in those domains with a higher potential (i.e., achieve more benefits with less risks).
3. Indication in absolute terms of the convenience to invest in a domain taking into account the organisation's attitude toward risk.

The analysis has been successfully applied in several European organisations, including small and big software development teams from diverse domains, including utilities, banking, control systems, etc.

The analysis is usually performed in a one-day workshop meeting involving participation of representatives from all the departments involved in the domain. It is specifically important to involve not only the software development department, but also systems and marketing/sales departments to bring a customer perspective to the analysis.

These workshop-type meetings have demonstrated to be extremely effective to achieve consensus across the different departments of stakeholders of a domain in a given organisation. When consensus is not reached, the differences among the participants are also recorded and shown in the graph. Actually the area associated with the domain is a representation of these differences and provides an indication of the uncertainty of the information collected.

The domain potential analysis is presented as a tool to facilitate the inclusion of a business perspective in setting a product-line strategy. It has been conceived with the idea of being simple enough to be usable, without requiring much effort or data that is not available in the organisations and, at the same time, complete enough to ensure a disciplined and repeatable analysis.

References

[Boehm 89] Barry W. Boehm, *Software Risk Management*, IEEE Computer Society Press Press, 1989.

[Brealey 91] Richard A. Brealey, Stewart C. Meyers, *Principles of Corporate Finance*, McGraw-Hill, 4th edition, 1991.

[CSE 93] Centre for Software Engineering, *Risk management for software development projects*, 1993.

[Favaro 98] Favaro J., *Value-based Reuse Investment*, Tutorial of the 2nd European Reuse Workshop, European Software Institute, Madrid, 1998.

[Kirkwood 97] Craig W. Kirkwood, *Notes on attitude toward risk taking and the exponential utility function* , January 1997.

[Lim 98] Wayne C. Lim, *Managing Software Reuse*, Prentice Hall, 1998.

[ODM 96] Lockheed Martin Tactical Defense Systems, *Organization Domain Modeling (ODM),* Guidebook version 2.0, 1996.

[Pike 96] Richard Pike and Bill Neale, *Corporate Finance and investment,* Prentice Hall, 1996.

[Poulin 97] Jeffrey S. Poulin, *Measuring Software Reuse*, Addison Wesley Longman, Inc., 1997.

[Withey 96] Withey J., *Investment Analysis of Software Assets for Product Lines*, Technical Report CMU/SEI-TR-010, Software Engineering Institute, Pittsburgh, PA, 1996.

Dependency Navigation in Product Lines Using XML[1]

Douglas Stuart, Wonhee Sull[2], and T. W. Cook

Microelectronics and Computer Technology Corporation (MCC)
3500 W. Balcones Center Dr.
Austin TX 78759, U.S.A.
{stuart, sull, tw}@mcc.com

Abstract. MCC's *Software and Systems Engineering Productivity* (SSEP) Project brings together emerging software engineering technology in architecture definition, analysis and generation, product family engineering, and rationale capture to enable organizations to achieve unprecedented levels of productivity and reuse. One of the central requirements for architecture-based product line development is the capability to efficiently and effectively navigate between related artifacts. This involves navigation both between product line artifacts and application artifacts, and between artifacts associated with successive development phases at each level. This navigation supports automated identification, selection and generation of artifacts, as well as manual development activities. This paper describes the Architecture Description Language (ADL) and toolset used by the SSEP project, and its support for linking and navigation in a product line development environment.

1. Introduction

MCC's *Software and Systems Engineering Productivity* (SSEP) Project brings together emerging software engineering technology in architecture definition, analysis and generation, product family engineering, and rationale capture to enable organizations to achieve unprecedented levels of productivity and reuse. An emphasis of the SSEP project is the development of tools supporting product family oriented development.

The current prototype toolset includes VisualADML, ScenarioManager, WinWin, LinkManager, DependencyChecker, and ComposeTool. VisualADML is an editor for product family architecture descriptions. ScenarioManager is a tool for capturing scenarios for operational requirements. LinkManager provides for linking and navigation between product family artifacts. DependencyChecker and ComposeTool are used to create applications in the product family from the product family architecture by first creating an application specific architecture and then building the product from the architecture and an asset base of component implementations.

---

[1] All company, product, and service names mentioned are used for identification purposes only, and may be registered trademarks, trademarks or service marks of their respective owners.

[2] This author is now at SK Telecom, Korea, and may be reached at sull@sktelecom.com. The work described in this paper was performed while the author was at MCC.

F. van der Linden (Ed.): IW-SAPF-3, LNCS 1951, pp. 82-93, 2000.

One of the central requirements for architecture-based product line development is the capability to efficiently and effectively navigate between related artifacts, in particular between artifacts with semantic dependencies. This involves navigation both between product line artifacts and application artifacts, and between artifacts associated with successive development phases at each level. This navigation supports automated identification, selection and generation of artifacts, as well as manual development activities.

This paper describes the toolset used by the SSEP project and its associated artifacts, in particular the Architecture Description Language (ADL), and its support for linking and navigation in a product line development environment.

2. ADML

ADML, Architecture Description Markup Language, is an architecture description language (ADL) [1] based on ACME [2] using XML [3], the Extensible Markup Language, as a representation language. ACME and XML both have specific advantages in the representation of software architectures for product families, in addition to their other features. The use of ADML as an architecture representation for the full life cycle of product family engineering capitalizes on these advantages.

The basic features of ADML are the basic features of ACME. The top level of an ADML specification is a *design.* A design is made up of systems. Each system represents an architecturally complete entity. A system is in turn made up of *components* and *connectors* related via *attachments*. As architecturally complete entities, systems can be used to represent applications within a product family, or using *representations*, as refinements of other architectural elements. The relationship between an architectural element and a system that refines it is specified using a *binding*. The abstraction facility provided by representations and bindings supports iterative refinement, implementation and design reuse, and the externalization of implementation of architectural elements, all of which are important to the successful application of product families.

Components represent the computational and data elements of an architecture. Components have *ports* that define their architectural interface. Connectors represent the interactions between components. Connectors have *roles* that define the participants in the interaction represented by the connector. One of the hallmarks of ADLs is that components and connectors are both treated as first class constructs.

The final aspect of ADML, and one particularly relevant to architectures for product families, is that each element in an ADML specification may have one or more properties associated with it. Properties are name-value pairs that have no native ADML semantics. Properties provide the basic extensibility mechanism of ADML. Although individual properties may have no ADML interpretation, ADML extensions may provide such properties with semantic interpretations. For example, the basic ADML interpretation of a property ("color", "red") of a component is just that a property exists with name "color" and value "red". However, a visualization tool could interpret the property and display the component in red. In fact, just such an approach was used to build a visual editor for ADML.

2.1. Extensibility

In addition to other expected benefits of using XML as a representation language for ADML[4], such as access to COTS (Commercial-Off-The-Shelf) XML tools, there are two features of XML that provide specific advantages for a product family architectural representation. The first of these is extensibility, which is magnified by the extensibility provided by ACME, the other foundation of ADML. As discussed above, the property list mechanism, inherited by ADML from ACME, provides a means for extending ADML semantically. The extension mechanisms of XML provide a means for extending ADML syntactically.

An XML DTD (Document Type Definition) provides a means for defining XML based languages. XML documents consist of elements, each of which may have attributes (another property mechanism), sub-elements, and content. DTDs define an XML based language as a context free grammar of attributed elements. DTDs provide one means of extension. The ADML DTD may be modified to include additional attributes and elements appropriate for a particular product family. For example, performance attributes could be added to component and connector elements in an ADML extension for a real-time product family. The DTD also provides a mechanism for extracting multiple views from a single architectural document. XML Architectures [5] provide a mechanism for projecting documents consistent with one DTD onto another DTD. XML documents can also be checked for conformance to DTDs, providing a way to enforce consistency in architectural specifications.

2.2. Linking

A second feature of XML (and associated standards) that supports architectures for product families is linking. The capability to define and navigate relationships among artifacts is essential to effective development of products in product families. The central notion in product family based development is that products in the family are related and exploiting the relationships will lead to more effective development of individual products. The types of relationships include refinement and implementation relationships. Such relationships include those between the requirements common to all members of the product family and the specific requirements on an individual product within the family, and between components in the product family architecture and their implementations in the product family reuse repository.

XML, among other features, provides for links with multiple locators (targets) into documents that may be represented external to the linked documents. These features support the sophisticated linking strategies needed for product families. For example, the links from a document in a legacy format may be stored external to that document facilitating use of legacy tools in a product family environment. In addition, a single link can be used to establish a relationship between a requirement and all of the architectural elements implementing that requirement.

The linking facilities of XML in particular support the entire product family lifecycle of both domain and application engineering. Links between product family requirements and product family architecture can be used to create an application architecture from the application requirements. Links between the application

architecture and the product family reuse repository can be used to automatically generate an application within the product family.

ADML exploits the linking capabilities of XML, and the extensibility of both XML and ACME, to provide a language for product family software architectures. ADML can be extended for each product family to provide the needed expressiveness. This may include extensions for particular architectural quality attributes relevant to the product family, for requirements traceability, for architectural design rationale, for architectural consistency, or for architectural element implementations. The demonstration described in the next section gives concrete examples of how ADML can be used to capture the architecture of a product family and generate the architecture of individual products within the family.

Having an infrastructure that supports links, the next issue that needs to be addressed by a product line development environment is the actual creation of links by developers. Although automatic link creation is the ideal, current understanding and tool support do not support it. For example, one approach to automatic link creation would rely on proximity of access to create links. For example, if two documents were simultaneously edited, then a link would be created between them. Although such links could be useful, such links are likely to be redundant and unhelpful, since human intervention is almost certain to be necessary to supply any semantics to such a link.

Pending the development of more advanced techniques, most links will be created manually by engineers, either domain engineers or application engineers. There are two approaches to manual link creation. Either integrate link creation mechanisms into each of the tools in the development environment, or provide a separate link creation tool. The first approach has the advantages of not introducing a new tool, and the potential for convenient or semi-automatic link creation. The second approach requires the developer to use a separate tool, increasing the perceived effort to create links, but such a tool is probably necessary in any event for maintenance and navigation of links, and is required if the primary development tool can not be extended to capture links. The second approach is followed in this investigation.

3. A Product Line Demonstration

A demonstration was performed to explore the utility of navigation approaches and tools. The demonstration took the form of creating a product line using the prototype SSEP Toolset. The toolset consists of a number of product line development tools that create artifacts and links, and that manually and automatically navigate links. Two applications were created within the product line. The tools were used to create a set of product line requirements and application requirements for the two applications, and an architecture for the product line. Links were also created between the product line and application requirements, between the requirements and the architecture, and between the architecture and a virtual asset base implementing the architectural elements. Manual navigation was explored by moving between both levels of requirements, and between requirements and architecture. Automatic navigation was explored by automatically creating application architectures from the product line architecture and an identified subset of requirements.

4. Linking in the SSEP Toolset

The introduction to this document described a number of potential applications for navigation between product line artifacts. This section refines those notions in the context of experiments carried out using elements of the prototype SSEP Toolset for architecture based product line development. The current prototype SSEP Toolset contains two tools that are used for requirements capture and domain modeling. WinWin [6] is a requirements negotiation and rationale capture tool developed at USC. The toolset also includes ScenarioManager, a tool for capturing scenarios in SML (Scenario Markup Language).

The version of WinWin in the prototype toolset has been modified to export RCML (Requirements Capture Markup Language), an XML based language for WinWin based requirements. Although RCML includes the full complement of WinWin artifact types, two artifact types are of special interest. Issues are used to represent requirements. Options are used to represent the variability of the product line [7] in a dimension represented by a requirement Option. Note that this usage differs from the usage of the corresponding WinWin artifacts in the WinWin process model.

The prototype SSEP Toolset also includes LinkManager, a tool for manually creating and navigating links between XML documents, in particular those that represent product line artifacts. The link manager maintains a list of product line XML documents, and can be used to create multi-way directional links between elements in any of the documents. The links are stored in an external link file so that the set of product line links may be modified independent of the underlying XML files. This is particularly important since WinWin can write but not read RCML.

In addition to creating links, LinkManager can also be used to navigate the created links. This can be done either by selecting a link directly from the list of links in the product line artifact base, or by selecting an element in any XML document in the product line artifact base and using LinkManager to find the links to or from the element, and then navigating the link. This basic functionality was sufficient to perform experiments to determine the utility and practicality of the various modes of navigation discussed in the introduction.

One type of navigation proposed is manual navigation between product line requirements and application requirements. In the experiments performed, the product line requirements were represented by an RCML document. The application requirements were represented by SML documents for each application that identified the particular choices within the product line variability. Links were created between the product line variability alternatives, represented by Options in the product line RCML document, and the corresponding Actions and Actors in an SML document representing a particular application within the product line. Since both product line and application requirements are captured in XML form, the links are also XML, making possible the creation of multi-way links. For example, the same product line requirement Option can be linked to several different application level SML Actions and Actors that realize the option with a singular link. This facilitates maintaining consistency since all of the application artifacts that are the realization of a single product line requirement are identified in a single artifact.

Automatic navigation was explored by creating tools that would automatically generate an application architecture from the product line architecture and a set of

application requirements. This involved two tools. The ACS (Architecture Components Selector) facility of LinkManager, when given a set of requirements artifacts, would automatically generate a set of product line architecture components by navigating the links between the application requirements (RCML Options) and the product line architecture. The set of selected components serves as the basis for the application architecture and is supplied to a second tool, DependencyChecker.

DependencyChecker uses the set of components supplied by ACS as a basis set for an application architecture. The product line architecture includes dependencies between components. If a component is present in an application architecture, any component that it has a dependency to must also be included. In the architecture of the experiment, such dependencies primarily reflected a delegation relationship. DependencyChecker creates an application architecture by adding to the architecture any components that components in the architecture depend on.

The success of LinkManager in providing manual link navigation and the ACS tool and DependencyChecker in exploiting automatic link navigation validate the utility of both types of navigation in product line development. Manual navigation was effective in moving between product line and application requirements and allowed both domain and application engineers access to the artifacts needed to perform product line development. Further, the use of XML greatly facilitated both types of navigation.

The experiments performed also highlighted several areas for both improvement and further experiment. First, there were only two types of dependency used in this experiment, the "requires" dependency between components in the product line architecture, and an implementation/refinement dependency between product line and application requirements. Further examination may yield other useful types or subtypes of dependency. For example, all of the dependencies between product line and application requirements in the experiment were AND implementation dependencies. That is, a particular product line requirement is implemented by the combination of all of the application requirements that it is linked to. It is easy to see that OR implementation dependencies might also be useful. There may be other types of dependencies that could be identified.

Another capability provided by the SSEP Toolset due to the use of XML is the ability to have links to links. There are a number of ways such links could be used to support product line development. One possible use is to have a single link representing an application within the product line that is linked to all of the individual product line requirements to application requirements links. Such a link-to-links could be used to rapidly identify an application. Further applications of such links could also arise.

5. Example

The following simple example illustrates the different types of links and approaches to link navigation currently provided by the MCC toolset. The file extracts included in this example are not the complete XML files generated by the toolset. The complete files for this example are not included in the paper both in order to highlight salient aspects and to conserve space.

Consider a hypothetical flatbed scanner software product line. Product line scoping and domain analysis has determined that there are two relevant scenarios for using flatbed scanners. They are used either to scan pages containing text that will be extracted from the scanned image and saved in a text file using some standard text file format, or to scan pages containing a photograph that will be saved in an image file using some standard image file format. These two scenarios are captured using ScenarioManager and saved in an SML file, an extract from which appears in Figure 1.

```
<ScenariosDocument>
    <Agents>
      <Agent name="image capturer" />
      <Agent name="photo processor" />
      <Agent name="text processor" />
    <Actions>
      <Action name="capture image"
        <AgentRef href="# (image capturer)" />
      <Action name="process text image"
        <AgentRef href="# (text processor)" />
      <Action name="save text file" >
        <AgentRef href="# (text processor)" />
      <Action name="process photographic image" >
        <AgentRef href="# (photo processor)" />
      <Action name="save photo file" >
        <AgentRef href="# (photo processor)" />
    <Scenarios>
      <Scenario name="text image">
            <ActionRef href="# (capture image)" />
            <ActionRef href="# (process text image)" />
            <ActionRef href="# (save text file)" />
      <Scenario name="photographic image">
            <ActionRef href="# (capture image)" />
            <ActionRef "# (process photographic image)" />
            <ActionRef "# (save photo file)" />
```

Fig. 1. Scenario SML File

Examining the two scenarios, the domain engineer determines that the variability in the product line resides in the type of page being scanned. The product line will support either scanning photographs or scanning text. WinWin is used to capture this product line variability in the RCML file in Figure 2 using the WinWin artifact types *agreement*, *issue*, and *option*. Note that there are two products in the product line, one represented by the agreement *text scanner* and the other by the agreement *photo scanner*. The first selects the option *text image* from the variability represented by the issue *image type*, while the second selects the photo *image option*.

```
<Project Name="scanner">
  <Users>
    <ProjectUser Name="domain" />
  <Artifacts>
   <Issue identifier="domain-ISSU-1" name="image type">
    <Agreement identifier="domain-AGRE-1"
          name="text scanner" >
      <ArtifactBody> Scanner that processes text pages
    <Agreement identifier="domain-AGRE-2"
          name="photo scanner" >
      <ArtifactBody> Scanner that processes
           photographic pages.
    <Option identifier="domain-OPTN-1"
          name="text image" >
      <ArtifactBody>Captured image is a treated as text
    <Option identifier="domain-OPTN-2"
          name="photo image" >
      <ArtifactBody> Captured image is a photograph
```

Fig. 2. WinWin RCML file

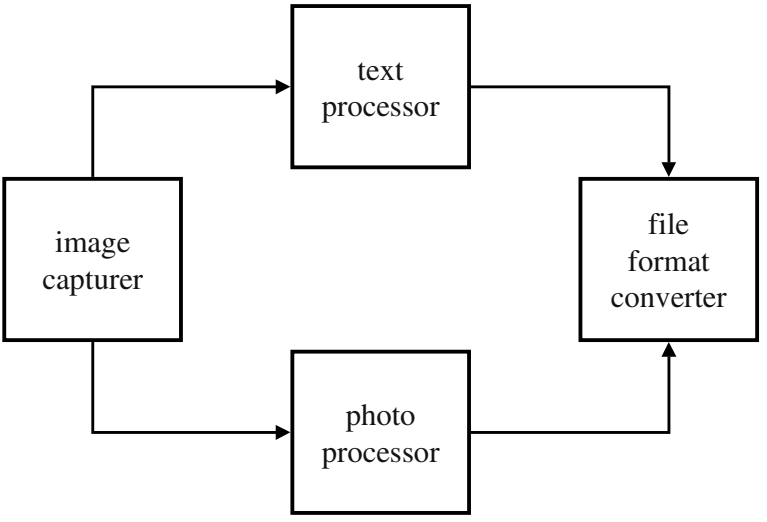


Fig. 3. Reference Architecture

With the domain model defined by the scenarios and planned variability, the domain engineer now creates the product line reference architecture. In doing so, the domain engineer identifies two elements of commonality within the product line. The same image capture component can be used regardless of the type of image being captured, and an existing file conversion component can be used to convert between different standard text file formats and between different standard photographic file formats. This leads to the reference architecture shown graphically in Figure 3 and as an ADML file extract in Figure 4. The ADML file includes two *Dependency* properties that indicate both image processing components require the *file format converter* component.

```
<ADMLDesign>
  <SystemDeclaration  name="Scanner">
    <ComponentDeclaration  name="image capturer">
    <ComponentDeclaration  name="text processor">
      <SequenceDeclaration name="Dependency">
          <LiteralDeclaration type="String"
           value="file format converter" />
    <ComponentDeclaration name="photo processor">
      <SequenceDeclaration name="Dependency">
          <LiteralDeclaration type="String"
           value="file format converter" />
    <ComponentDeclaration name="file format converter">
```

Fig. 4. Reference architecture ADML file

As the activities described above are carried out using ScenarioManager, WinWin, and VisualADML, the domain engineer will also be creating links between related artifacts using LinkManager. An extract from the final link file is presented in Figure 5. Note that there are links between the RCML file agreements representing product instances within the product line and the corresponding SML file scenarios. Also, there are links between the agents that carry out the actions in the scenario and the components in the product line architecture ADML file that implement them.

```
<LinkedDocuments>
  <Links name="default_links" title="Links">
    <Link name="text-os" direction="oneway">
          <Locator
href="scanner_RCML.xml#identifier(domain-OPTN-1)" />
          <Locator href="Scenarios.sml#(text image)" />
    </Link>
    <Link name="photo-os" direction="oneway">
          <Locator
href="scanner_RCML.xml#identifier(domain-OPTN-2)" />
          <Locator
href="Scenarios.sml#(photographic image)" />
    </Link>
    <Link name="capture-agentcomp" direction="oneway">
          <Locator
```

```
href="Scenarios.sml#(image capturer)" />
          <Locator
href="scannerpla.adml#id(image capturer)" />
     </Link>
     <Link name="photo-agentcomp" direction="oneway">
          <Locator
href="Scenarios.sml#(photo processor)" />
          <Locator
href="scannerpla.adml#id(photo processor)" />
     </Link>
     <Link name="text-agentcomp" direction="oneway">>
          <Locator
href="Scenarios.sml#descendant(1,Agent,name,text
processor)" />
          <Locator
href="scannerpla.adml#id(text processor)" />
     </Link>
     <Link name="text-agop" direction="oneway">
          <Locator
href="scanner_RCML.xml#identifier(domain-AGRE-1)" />
          <Locator
href="scanner_RCML.xml#identifier(domain-OPTN-1)" />>
     </Link>
     <Link name="photo-agop" direction="oneway">
          <Locator
href="scanner_RCML.xml#identifier(domain-AGRE-2)" />
          <Locator
href="scanner_RCML.xml#identifier(domain-OPTN-2)" />
     </Link>
     <Link name="s1-scag" direction="oneway">
          <Locator
href="Scenarios.sml#(text image)" />
          <Locator
href="Scenarios.sml#(image capturer)" />
          <Locator
href="Scenarios.sml#(text processor)" />
     </Link>
     <Link name="s2-scag" direction="oneway">
          <Locator
href="Scenarios.sml#(photographic image)" />
          <Locator
href="Scenarios.sml#(image capturer)" />
          <Locator
href="Scenarios.sml#(photo processor)" />
     </Link>
   </Links>
</LinkedDocuments>
```

Fig. 5. Links file

When an application engineer creates an individual application within the product line, LinkManager is used to navigate the links between artifacts created by the domain engineer. As discussed above, this navigation can be either manual or automated. As an example of manual navigation, the application engineer learning about the nature of the variability in the product line can follow the link from the RCML file *text image* option (*domain-OPTN-1*) to the corresponding *text image* scenario in the SML file. As an example of automated navigation, to create the text imaging application, the application engineer can select the *text scanner* agreement element in the RCML file and invoke ACS. ACS will then automatically traverse the links in the product line links file from the agreement to the corresponding scenario, to the actions that make up the scenario and their corresponding agents in order to identify the components required for the application instance. In this case, selecting the *image capturer* and *text processor* components. DependencyChecker could then be used to generate the complete instance architecture by adding the *file format converter* due to the ADML file dependency of the *text processor* component.

6. Architectural Analysis

An area for future study is the use of dependency information in the analysis of product lines and applications within the product line. The promise for early analysis provided by dependency navigation is evident in the DependencyChecker tool. Although currently configured to generate consistent architectures, ones in which each dependency is satisfied, it can also be used to determine if an architecture supplied as input is complete with respect to its dependencies. A richer dependency language would provide expanded scope for such analysis.

Likewise, navigation between product line and application requirements provides a basis for automatic generation of test suites for applications within the product line. The test artifacts associated with particular product line requirements can be identified by navigating dependencies from the application requirements to the related product line requirements to the appropriate test suites. The test suites thus constructed would form a basis for testing the application. Note, too, that if test suites were associated with domain model, product line architecture, or product line component implementation artifacts, such test suites could also be identified by navigating the appropriate dependencies. The automatic creation of a basis set of test suites would help ensure adequate test coverage for applications within the product line. Note that additional test suites corresponding to unique requirements of the application would have to be separately created.

7. Conclusion

Navigation between the various artifacts that arise during architecture based product line development is crucial to successful product line development. The goals of product line development are achieved through planned reuse of product line artifacts in multiple applications in the product line. Such strategic reuse depends on the application engineer being able to develop the application within the context of the

product line, requiring access to product line artifacts in context. Such in context access can only be provided by links capturing the dependence relationships between product line artifacts. This document has explored the utility of such artifacts, and documented the viability of their use through an experiment. Although the dependency based navigation between product line artifacts described here is valuable, there are a number of potential extensions to the basic navigation capabilities described here that are worthy of further investigation.

References

1 Neno Medvidovic and Richard N. Taylor. A framework for classifying and comparing architecture description languages. In Proceedings of the Sixth European Software Engineering Conference together with Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering, pages 60-76, Zurich, Switzerland, September 1997.

2 David Garlan, Robert Monroe, and David Wile. ACME: An architecture description interchange language. In Proceedings of CASCON 97, November1997.

3 Lars Marius Garshol. Introduction to XML, http://www.stud.ifi.uio.no/~larsga/download/xml/xml_eng.html, May 1998.

4 Steve Pruitt, Doug Stuart, Wonhee Sull, and T.W. Cook, The Merit of XML as an Architecture Description Language Meta-Language, Position Paper for WICSA, San Antonio, Feb. 1999.

5 Eliot Kimber. A tutorial introduction to sgml architectures. Technical report, ISOGEN International Corp., 1997.

6 Barry Boehm, Prasanta Bose, Ellis Horowitz, and Ming June Lee. Software requirements negotiation and renegotiation aids: A theory-w based spiral approach. In Proceedings of the 17th International Conference on Software Engineering (ICSE-17), Seattle, April 1995.

7 Wonhee Sull. Investigation on variability in product line engineering (In Preparation) 1999.

Summary of Product Family Concepts Session

Juha Kuusela[1] and Jan Bosch[2]

[1]Nokia NRC
Itämerenkatu 11-13
00180 Helsinki, Finland
[2]University of Karlskrona/Ronneby
S-372 25 Ronneby, Sweden
Juha.Kuusela@research.nokia.com, Jan.Bosch@ipd.hk-r.se

Introduction

The notion of software product families is still hard to control and manage. One can identify a number of reasons for this. First, the different dimensions of variation within a family result in, among others, overlapping feature interaction problems. Second, the relationships between a product and the family is not always trivial.

The title suggests this session was concerned with the concepts that underlie software product lines. Interestingly enough, the papers in the session actually did not address this, but rather discussed a number of instances of or approaches to representing software architecture concepts.

The questions that we asked were

- How to control and manage different dimensions of variation within families?
- What is the relationship between products and families?
- How to design on a family level?
- How to support instantiation of products, refinement of architectures and traceability of requirements and their changes.
- How to define connectors, components, reference architectures?

These questions should be answered so that we can also see how different documents developed during PL process should be structured and how each one of these artifacts are linked to each other so that we can navigate from one to another. This session emphasize was on practical solutions to these questions. Solutions that work even if we do not have universal semantics integrating all models.

Papers

First paper in the session was *Dependency Navigating in Product Lines Using XML* presented by Douglas Stuart. He explained that MCC's SSEP project is focusing on tools to support architecture based PL development. Their approach is based on cleverly combining existing tools and their own development.

As an architecture representation they use ACME based ADL. The real clue behind the tool set is architecture description markup language (ADML) based both on ACME and XML. Use of XML brings COTS support for parsers, browsers, editors and an ability to flexibly link structured documents (n-to-n, external).

First generation of this tool set is now complete and one case study has been completed and support for dependency based navigation between product line artifacts has proven valuable. The work continues to develop a new ADML editor,

F. van der Linden (Ed.): IW-SAPF-3, LNCS 1951, pp. 94-95, 2000.

Jini based implementation of Link Manager and further development of ADML (AML, Analysis).

Second paper was *Software Connectors and Refinement in Family Architectures* presented by Alexander Egyed. This paper emphasizes the role of generic software connectors in supporting flexible product families. Role of these connectors is to mediate interactions among components, to provide auxiliary mechanisms for interaction, and to define dependencies and protocols.

Authors also claimed that the connectors are largely application domain independent and that the types of connectors are far limited than the types of components. Connectors could also isolate certain system properties like deployment profile, concurrency, security and reliability. Question *is industry willing to standardize connectors* was raised.

Bounded connectors can support analysis of family architecture even if components are still ambiguous. Modeling variation in components can be done through connectors. Paper also introduces family designs as a mechanism to avoid problems of architecture refinement and bridging the gap between product architecture and design.

This paper made strong conclusions like connectors are better in modeling bounded ambiguity, connectors are more flexible (e.g. COTS), architectural modeling requires consistent refinement and evolution, family designs can be used as intermediate models.

The third presentation *System family architectures: current challenges at Nokia* was given by Alessandro Maccari. He explained the complexity of feature variation in mobile phones coming from network standards, handset categories, user interfaces, operating systems and country-specific requirements.

Inn this situation it is not clear how to do requirements management and engineering, how to structure software with variance and how to best model product families. Epoc based smart phone family is the current challenge where application of family architecturing techniques can provide real impact. These phone are highly complex but the family includes only few products.

Discussion and Observation

The role of connectors was discussed. Their role as suggested by the second paper was not really accepted. Participants pointed out that variation has to be modeled both on connectors and components. Also the stability of connectors and components may vary. Sometimes connectors are just the glue code needed to integrate existing components.

Mainly discussion concentrate on the role and nature of features. What they are, how they can be manipulated and what is the role of features in scoping and defining a product line architecture.

The role of navigation aids in supporting traceability of requirements and impact analysis of changes was also touched.

When the session ended we still had more questions than answers.

Software Connectors and Refinement in Family Architectures

Alexander Egyed, Nikunj Mehta, and Nenad Medvidovíc

Department of Computer Science
University of Southern California
Los Angeles, CA 90089-0781, USA
{aegyed, mehta, neno}@sunset.usc.edu

Abstract. Product families promote reuse of software artifacts such as architectures, designs and implementations. Product family architectures are difficult to create due to the need to support variations. Traditional approaches emphasize the identification and description of generic components, which makes it difficult to support variations among products. This paper presents an approach to modeling family architectures using generic software connectors that provide bounded ambiguity and support flexible product families. The paper also proposes an approach for transforming a family architecture to a product design through a four-way refinement and evolution process.

Introduction

Large, complex systems are often developed in the context of *product families*[1] . This enables developers to maximize reuse, accelerate the development process while reducing costs, and deliver products that are generally more reliable. Reuse across product families occurs in terms of architecture, design and implementation. Architectural idioms identify the kinds of building blocks that may be used to compose a system and specify the constraints on the way the composition is done. An explicit focus on common *architectural* idioms has the potential to fundamentally transform the nature of software development, as component *integration* replaces implementation as the predominant development activity. The promise of software architectures is that better software systems can be built in this manner more quickly by modeling their important aspects throughout, and especially early in the development. Coupling the benefits of product family-based and architecture-based development has been recognized as an area with a great potential payoff, as evidenced by a growing number of conferences, workshops, and symposia that focus on this subject [2, 3, 4, 8, 12].

The existing body of research in the area of software architectures for product families is characterized by two major foci:

[1] In this paper, we use the following phrases interchangeably: families, application families, product families, product lines, and domain-specific software.

F. van der Linden (Ed.): IW-SAPF-3, LNCS 1951, pp. 96-106, 2000.

1. specification of generic, product family architectures (also referred to as *reference* architectures) and their instantiation into application architectures (e.g., [11]); and
2. identification and integration of reusable components that comprise different members of a product family (e.g., [5]).

In this paper, we focus on two additional issues that have not been addressed by existing approaches and that are useful complements to those identified above:

1. the role of software connectors in specifying and ensuring the extra-functional properties of both a product family and individual applications within the family; and
2. refinement of an instantiated product architecture into a design and, eventually, an implementation.

The role of *connectors* in software architectures is to isolate all communication, coordination, and mediation [10]. Connectors do not generally provide domain-specific functionality, but rather enable and streamline interactions among the functional elements (components). Thus, our hypothesis is that certain varying properties of applications within a family (e.g., deployment profile, concurrency, interoperability platform, performance, reliability, security, etc.) can be isolated within connectors. Also, certain types of connectors may occur regularly within a family. Our on-going work on classifying software connectors will serve as a vehicle for exploring these issues.

To enable the *refinement* of an architecture into its implementation, we leverage our work on transforming architecture-level constructs (specified in an architecture description language, or ADL [9]), into design-level constructs (specified in the Unified Modeling Language, or UML [6]), and enabling the refinement of the resulting high-level design in a property-preserving manner [1, 7]. We introduce the notion of *product family design*, analogous at the design level to a product family architecture. A product family design captures recurring design patterns across components in a family. Another hypothesis is that both product family designs and product architectures are needed to enable effective refinement.

The paper is organized as follows. Section 2 identifies the relationships between products and families, and between architectures and designs. Section 3 outlines the role of software connectors in family architectures. Instantiation of a product family architecture and refinement of the resulting product architecture into its implementation is discussed in more detail in Section 4. Section 5 presents an example illustrating the approach. Conclusions and a discussion of open issues round out the paper.

Relationship of Products and Families

Software architectures can be described using components, connectors and configurations [9]. Components are units of data store or computation whereas, connectors model the interactions among components. Architectural description identifies the obligations and freedoms of a software system built to that architecture. Obligations allow a high level analysis of system properties while the freedoms allow developers to design and implement the system according to the characteristics and constraints of an underlying infrastructure. Since a product family consists of products with commonalities and differences, it is useful to capture these aspects of

the individual products in *family architecture*. Moreover, use of the same architectural elements to describe family architectures and individual product architectures aids in keeping these artifacts consistent and simplifies understanding.

A family architecture provides generic information common to all the products of the family. This common information may include features present in all systems or a list of possible alternatives that products exhibit. It is easy to represent family architectures in terms of the similarities alone. However, in order to support variations in the individual products, a family architecture needs to describe the architectural elements with a certain amount of ambiguity. The product architecture, on the other hand, identifies specific architectural choices for a single product and thus can be considered as an instantiation of the family architecture. The product architecture is less ambiguous since all architectural elements are already chosen and specified for the sake of completeness. Proceeding in another direction, the family architecture can also be refined to create a set of more detailed family designs that can be used in individual products to obtain the recurring functional and extra-functional properties.

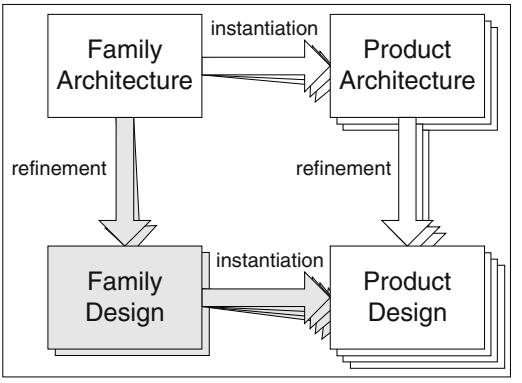


Fig. 1. Design refinement and instantiation using product architecture and family design

Figure 1 depicts a high-level framework we propose for architectural modeling of product families and their improved refinement and evolution. The white boxes and arrows in the figure denote the traditional way of instantiating product architectures from family architectures, followed by the refinement of product architectures into their designs (and subsequent implementations). To complement this traditional approach, we introduce the concept of *family design*. A family design contains design-related information about a product family that the architecture did not (or could not) specify. For instance, a family design could contain different design interpretations of architectural elements (e.g., in the form of design patterns). Merging the product architecture information with family design information can then lead to a product design. Therefore, the product architecture defines at a high level *what* needs to be designed and the family design provides information on *how* to design it. This four-way relationship between architecture, design, family, and product implies that there are at least two alternate but complementary paths of creating designs for the products of a given family.

Software Connectors in Family Architecture

Family architectures capture the essential properties and relations of the product family. They describe structural and behavioral freedom and model the functional and extra-functional aspects of a family. Behavioral freedom and functional aspects of the product family are typically captured in components. Our approach also supports the description and analysis of extra-functional properties, coupled with the identification of the structural freedoms through the use of semantically rich connectors.

As discussed above, family architectures need to describe commonalities as well as variations among family members. Commonalities can be captured through elements that are mandatory to all products. The real difficulty lies in modeling variations that have to be supported at the level of a product family. Various approaches have been proposed to describe family architectures including the use of styles, parameters, constraints and service provisioning. However, as discussed in [11], none of these techniques alone adequately addresses the problem of supporting variations in the product family. There is clearly a need for defining family architectures with a certain degree of *bounded ambiguity* in order to support product variations.

Consider the case of a customer service product family that needs to support two product domains, retail banking and telephony. These products require variations in terms of the underlying information, as well as in the interaction of the architectural components. The banking application requires online transactions, whereas the telephony product requires a batch update. A useful family architecture would be able to support the description of both kinds of products.

Software architecture captures the essential structural and behavioral information in the form of components and connectors. Family architectures are useful because they lead to better structured reuse and also because the bulk of the architectural analysis can be performed at the level of an entire family. There is a tradeoff between vagueness of description and the scope of applicability when it comes to specifying the architecture in product families. At the level of a product family, components tend to be vaguely described because family architectures need to support a variety of product features. This vagueness about components reduces our ability to reason about the family architecture. On the other hand, in the product architecture, components are described more precisely [13]. This tradeoff gives rise to reduced analyzability at the family architecture level and reduced flexibility of the concrete products.

Current techniques for representing components in family architectures, however, tend to be inflexible. Many reuse techniques depend on the availability of interchangeable components that can lead to a component marketplace. However, experience shows that such components can only be achieved through considerable standardization efforts. Standardization tends to be a long process in which decisions are often made at a corporate level rather than industry-wide level. Component centered reuse therefore tends to take longer to adapt and is applicable only to niche domains. On the other hand, the software industry has very quickly embraced component integration frameworks such as DCE RPC [14], COM [15], CORBA[16] and Enterprise Java Beans [17]. This indicates that the industry is more amenable to accepting standards of integrating components than to standards of defining components. We therefore focus our research on the role of software connectors in family architectures.

Software connectors describe the interactions among architectural components and support communication, coordination, conversion and facilitation needs of components [18]. Connectors can be used to describe interactions among components in family architectures. Furthermore, many extra functional properties of a system can be attributed to semantically rich connector mechanisms such as events, distributors and arbitrators. Since connectors can be applied across problem domains, they have a high potential for reusability. Connectors also significantly affect global system properties such as availability, throughput, security and scalability. Various architectural-styles motivated by software connectors have been studied, e.g. pipe and filter [19], real-time data feeds [20], event-driven architecture [21], message-based style [22], middleware-induced styles [23], and push-based systems [11]. Architectural styles are an important mechanism for enabling reuse in family architectures [11, 24], indicating that software connectors have a major role to play in enabling architecture-based reuse.

Connectors provide bounded ambiguity that is necessary for supporting variations in family architectures. In order to effectively exploit that ambiguity we have used a taxonomy view of software connectors that describes the connector types, dimensions and their possible values (see Figure 2 for an extract). The ambiguity is contained in the various dimensions along which a connector can be characterized and the range of values that a connector can assume for each dimension. Since there are a finite number of values that can be assigned for each connector dimension, ambiguity involved in defining connectors in an architecture is bounded. Family architectures can be vague about the component interactions and as such can be described using imprecise connectors, i.e. connectors that identify a range of values for connector dimensions. Our taxonomy allows architects to choose the concrete connectors necessary to support interaction among components and to provide the dimensions along which each specific product can choose a different variation of interaction. Many extra-functional properties can thus be evaluated based on connectors as their dimensions of variation are known.

As a solution for the example problem introduced above, the customer service product family architecture would describe the required component interactions in the form of an event connector that allows variations along dimensions shown in Figure 2. It then becomes possible to describe both forms of required interactions - online transactions and batch updates - based on the event dimensions of *notification* and *synchronicity*. It is possible to describe the distribution profile of the interactions using the distributor dimensions *delivery* and *addressing*.

We are currently developing an infrastructure for using and experimenting with connectors for implicit invocation, real time communication and parallel execution. This infrastructure builds upon our previous work with event connectors and adaptors [22].

Four-Way Refinement and Evolution

Having discussed an approach to exploiting connectors in modeling family architectures and instantiating them into product architectures, we now discuss how to refine the resulting product architectures into individual product designs. A product architecture constitutes an effective milestone [25] for any project since it can be

analyzed and simulated to ensure the presence (or absence) of properties of interest. Nevertheless, it is still a difficult task to refine those architectural models into designs and actual implementations.

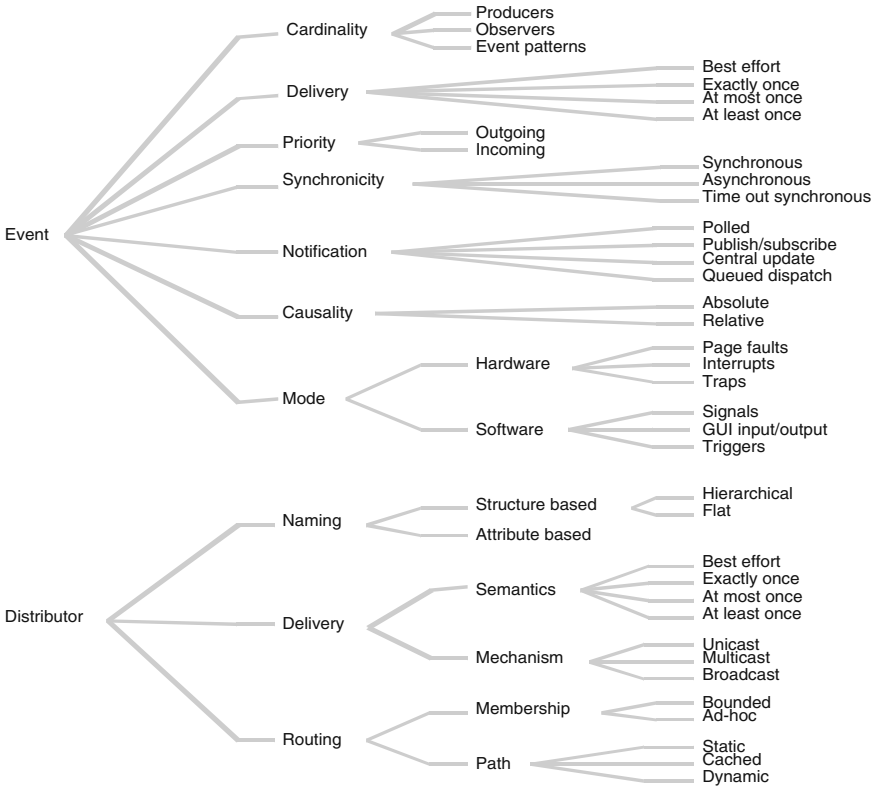


Fig. 2. An excerpt from the connector taxonomy showing (from left to right) types, dimensions, subdimensions, and values.

Refinement involves the creation of lower-level design models (and ultimately source code) and their continuous validation to ensure consistency. Refinement is difficult and it often has to be done manually. This, however, implies that defects may be introduced while refining the product design from its architecture. Thus, we are faced with a major problem: we create the family and product architectures with the understanding that they describe certain desirable properties the end system should exhibit; at the same time, if consistent refinement and evolution cannot be ensured, then there is no guarantee that the final product will indeed exhibit those properties. In other words, inconsistent refinement invalidates the purpose and utility of familiy and the product architecture. Meaningful architectural modeling *must* therefore ensure faithful refinement and evolution. This section will discuss an approach to improve the integrity of product models through the automation of refinement.

Automation during Refinement

The traditional way of modeling family architectures involves instantiating a family architecture into a product architecture, followed by refining that product architecture into a product design (Figure 3a – white area). This process may seem simpler in comparison to our proposed approach (Figure 3b – gray area) of using a family design, mainly because our approach additionally requires (1) modeling of a family design and (2) knowledge of how to relate it to the product architecture. However, we believe that these two additional activities ultimately simplify the overall refinement process. Using the traditional "family architecture to product architecture to product design" approach requires an instantiation from family architecture to product architecture and a refinement from the product architecture to a corresponding product design. The instantiation is relatively easy to do compared to the refinement activity, which is complicated by the lack of automation support. Even if the refinement could be automated, we would still be faced with the possibility of mismatch introduction at a later stage, e.g., when either the design or architecture is altered and those modifications are not properly propagated throughout the family.

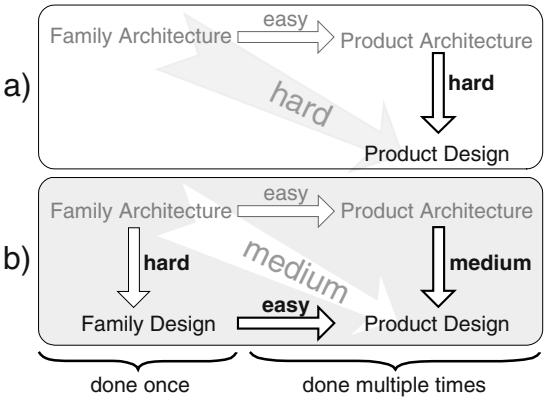


Fig. 3. Two refinement approaches.

Using a family design requires one additional instantiation activity – from a family design to a product design – which is again relatively straightforward. However, by replacing some (hard) refinement activities with (easy) instantiation activities, we achieve the added benefit of having to do less refinement, which, in turn, significantly simplifies doing product designs. Furthermore, we get the benefits of design reuse which complements architectural reuse (enabled through the use of family architecture). On the downside, arriving at a family design is not trivial or easily automated. The major advantage in using a family design is that we need to create it only *once* for each product family. Thus, once the family design is in place, each additional product can be architected and designed much more easily (because of having simplified the refinement of product architectures). On the other hand, without a family design we may avoid the hard initial task of creating and instantiating it, but the task of refining product architecture would be harder. It is not difficult to see the

return on investment of doing a one-time difficult task that simplifies a later repetitive task as opposed to avoiding that hard initial task but complicating the repetitive one. Therefore, family design allows us to shift parts of the hard repetitive tasks from product architecture refinement to family architecture refinement.

An added benefit of standard family designs is that they can be used to realize different species of the same connector type. For example, design patterns for central dispatch, publish-subscribe and queued dispatch events can be described in the form of family designs that realize the event connector, and are eventually instantiated in the product designs, based on the specific mode of interaction required. Additional design patterns can describe the other dimensions. This technique requires integration of design patterns in the family design for specific values of the different connector dimensions into a single software product design.

Continuing our previous example, the use of events in the customer support product family architecture as the means of component interaction would leave a lot of flexibility in the design of individual products; the family architecture can support a large number of variations in each product. The product architecture can be used to instantiate an event connector by selecting the dimensions of each connector instance in the family architecture. This is an easy step, as it would involve looking up the taxonomy of connectors and making choices for the dimensions of a connector. Family designs can then provide standard refinements in the form of design patterns of event-based interaction for different platforms and middleware environments. Finally, the product design would select specific design patterns for the target environment and desired product properties of the system.

Example: Going from Family Architecture to Product Design

Figure 4 depicts a simple example on how to use generic connectors and family design concepts to generate a product design from a family architecture definition. The figure depicts a simple accounting family architecture (upper left) that supports access to accounting information via an event-based connector. This particular family architecture allows two types of interfaces, one for ATM machines and one for terminal consoles, but only one at a time. The family design (lower left) depicts possible realizations of above architectural elements. Note that there are realizations for both architectural components and connectors. Furthermore, we need to be able to deal with incomplete family design specifications including missing links (e.g., missing glue code) and missing realizations for some architectural elements. For instance, *Flat File* is a realization of *Account*; however, it does not work together with any realization of *Event Bus* (*publish-subscribe*, *control dispatch*, or *batch update*).

In our example, we decided to instantiate a product architecture that consists of *Account*, *Event Bus*, and *Console Manager* (upper right). Using this product architecture as a reference and the family design as a resource database, we can now design and build the product using a predefined set of realizations. For instance, we could use the *Console PC Mgr* and combine it with either a *Publish-Subscribe* bus and a *Database*, or a *Batch-Update* bus and a *Flat File*. When we specified the product architecture, we also specified some attributes the architectural connectors should demonstrate. For instance, we pre-selected the *Event* bus to be of the *Batch Update* style. With this additional information, we can now automatically select a

possible product design from the family design that would be compatible with the product architecture (lower right). The product architecture supplies information on *what* to design, while the family design provides a details on *how* to design a product.

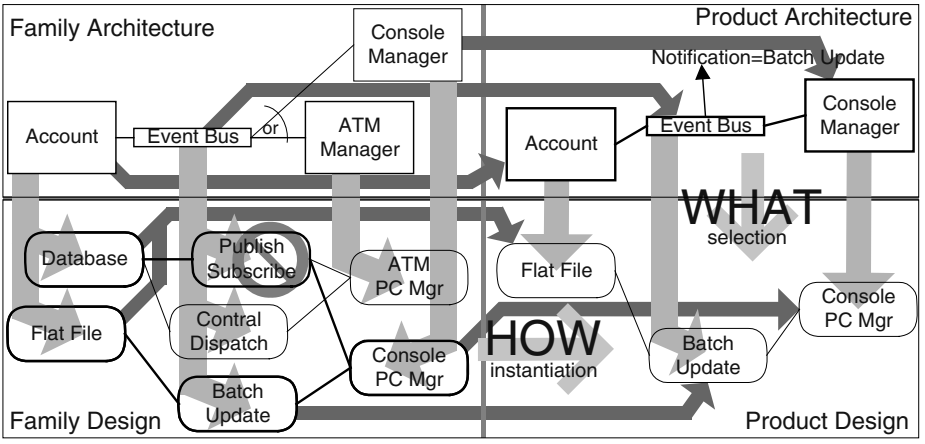


Fig. 4. Example of instantiation, refinement and traceability links

Conclusions

This paper identified and addressed two significant challenges in product family development: modeling family architectures via generic connectors and supporting automatic architectural refinement via family designs. Our approach involves the use of a taxonomy of connectors to model the bounded ambiguity in family architectures. We do not claim that connectors are more important than components for enabling family architectural descriptions; however we have found that, in some respect, connectors are significantly more flexible and reusable than components.

To enable automatic refinement and evolution, we introduced the concept of family design. Family designs provide a set of realizations of architectural components and connectors (e.g., in the form of design patterns). They simplify refinement by providing an additional path from a family architecture to a product design. We believe that combining a product architecture and a family design provides simplified and more precise refinement.

To date we have developed a suite of tools that allows automated mapping between architecture and design as well as their consistency checking. We have also studied the role of complex connectors in simplifying component integration and generating designs [1] and implementations [26]. The techniques used in this paper extend our previous work in the area of product line architectures [24]. This work is still in progress and it will evolve in several directions, including refining of our taxonomy of connectors, providing automated support for creating family designs, and resolving mismatches among architectural and design views at the level of a product family.

Acknowledgements

This research is sponsored by the Defense Advanced Research Projects Agency, and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement numbers F30602-94-C-0195 and F30602-99-C-0174, as well as by the Affiliates of the USC Center for Software Engineering. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Air Force Research Laboratory or the U.S. Government.

The authors are thankful to Sandeep Phadke and Marija Rakić who provided useful feedback while discussing the topics presented in this paper.

References

[1] M. Abi-Antoun and N. Medvidovic. Enabling the Refinement of a Software Architecture into a Design. In Proceedings of The Second International Conference on The Unified Modeling Language (UML'99), Fort Collins, CO, October 1999.

[2] ARES. Proceedings of the International Workshop on Development and Evolution of Software Architectures for Product Families, Las Navas del Marqués, Ávila, Spain, November 1996. http://hpv17.infosys.tuwien.ac.at/Projects/ARES/public/AWS/

[3] ARES II. F. van der Linden, editor. Proceedings of the Second International Workshop on Development and Evolution of Software Architectures for Product Families, Las Palmas de Gran Canaria, Spain, February 1998.

[4] ARES III. The Third International Workshop on Development and Evolution of Software Architectures for Product Families, Las Palmas de Gran Canaria, Spain, February 2000.

[5] Batory, L. Coglianese, S. Shafer, and W. Tracz. The ADAGE Avionics Reference Architecture. In Proceedings of AIAA Computing in Aerospace 10, San Antonio, 1995.

[6] Booch, I. Jacobson, and J. Rumbaugh. The Unified Modeling Language User Guide. Addison-Wesley, 1998.

[7] A. Egyed and N. Medvidovic. A Formal Approach to Heterogeneous Software Modeling. Alexander Egyed and Nenad Medvidovic, to appear in Proceedings of Foundational Aspects of Software Engineering, Berlin, Germany, 2000.

[8] R. Hayes-Roth and W. Tracz. DSSA Tool Requirements for Key Process Functions. ADAGE Technical Report, ADAGE-IBM-93-13B, October 1994.

[9] N. Medvidovic and R.N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. Accepted for publication in IEEE Transactions on Software Engineering, 2000. (To appear)

[10] D. E. Perry. Software Architecture and its Relevance to Software Engineering, Invited Talk. Second International Conference on Coordination Models and Languages (COORD '97), Berlin, Germany, September 1997.
[11] D. E. Perry. Generic Descriptions for Product Line Architectures. In Proceedings of the Second International Workshop on Development and Evolution of Software Architectures for Product Families (ARES II), Las Palmas de Gran Canaria, Spain, February 1998.
[12] The First Software Product Line Conference, August 28-31, 2000, Denver, Colorado, USA. http://www.sei.cmu.edu/plp/conf/SPLC.html
[13] D. Batory and S. O'Malley. The Design and Implementation of Hierarchical Software Systems with Reusable Components. ACM Transactions on Software Engineering and Methodology, 1(4), October 1992, pp. 355-398.
[14] The Open Group, http://www.opengroup.org
[15] Microsoft Corp. http://www.microsoft.com/com
[16] Object Management Group, http://www.omg.org
[17] Sun Microsystems. http://java.sun.com/j2ee
[18] N. Mehta, N. Medvidovic and S. Phadke, Towards a Taxonomy of Software Connectors, Technical Report, Center for Software Engineering, University of Southern California, USC-CSE-99-529, 1999.
[19] M. Shaw and D. Garlan. Software Architecture: Perspectives on an Emerging Discipline, Prentice-Hall, Upper Saddle River, NJ, 1996.
[20] N. Roodyn and W. Emmerich. An Architecural Style for Multiple Real-Time Data Feeds. 21st International Conference on Software Engineering (ICSE '99), Los Angeles, CA, May 1999.
[21] Carzaniga, E. Di Nitto, D. S. Rosenbloom and A. L. Wolf. Issues in Supporting Event-based Architectural Styles. 3rd International Software Architecture Workshop (ISAW3), Orlando FL, 1998.
[22] R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead and J. E. Robbins. A component- and message-based architectural style for GUI software. IEEE Transactions on Software Engineering, 1996, 22(6), pp. 390-406.
[23] E. Di Nitto and D. Rosenbloom. Exploiting ADLs to Specify Architectural Styles Induced by Middleware Infrastructures. 21st International Conference on Software Engineering (ICSE '99), Los Angeles, CA, May 1999.
[24] N. Medvidovic and R. N. Taylor. Exploiting architectural style to develop a family of applications. In IEE Proceedings Software Engineering, Vol. 144 No 5-6, October 1997.
[25] B. Boehm, Anchoring the Software Process, IEEE Software, July 1996.
[26] N. Medvidovic, D. S. Rosenblum, and R. N. Taylor. A Language and Environment for Architecture-Based Software Development and Evolution. In Proceedings of the 21st International Conference on Software Engineering (ICSE'99), pp. 44-53, Los Angeles, CA, May 16-22, 1999.

System Family Architectures: Current Challenges at Nokia

Alessandro Maccari and Antti-Pekka Tuovinen

Nokia Research Center
P. O. Box 407
FIN 00045 – NOKIA GROUP
Fax: +358 9 4376 6308
{alessandro.maccari,antti-pekka.tuovinen}@nokia.com

Abstract. We discuss the current state and the future challenges of software architecture work pertaining to the mobile handset families produced by Nokia. We identify the most important variance factors in the product set, present some open problems in the product family development, and outline a case study for employing the system family technology developed in the ESAPS project to the development of mobile handsets at Nokia.

Introduction

A mobile phone is a highly software-intensive system. The effectiveness of the development of its software is vital in terms of the time and cost. Organizing products in families where the software components are reused across the whole family of products offers possibilities for significant economical gains. In such an environment, the central problem is managing the product variability and the architectural evolution implied by new requirements [Kuusela99].

The *system family* (or product-line) approach appears as a promising way to systematically manage the development of product variants in a cost-effective way. Recently, the ESAPS project (Engineering Software Architectures, Processes and Platforms for System Families) [ESAPS] was launched with the aim to introduce the system family approach to European organizations of various types and sizes. Nokia is one of the industrial partners of the project. As a part of ESAPS, this work aims at investigating the issues that pertain to the introduction of the system family approach in the development of mobile handsets at Nokia.

In the following, we first identify the variance factors of the Nokia phones and discuss the current state and some of the challenges of software architecting at Nokia focussing on system family related issues. Then, we present an outline for an industrial-level case study that tries to apply the system family approach at Nokia. In the case study, we will employ the techniques developed in ESAPS for system family engineering.

F. van der Linden (Ed.): IW-SAPF-3, LNCS 1951, pp. 107-115, 2000.

Issues in System Family Architecture of Nokia Products

The product set of Nokia is vast. Currently, more than a hundred different models are available in the market, and this number continuously increases (see [Nokiaweb] for the updated list).

It is not difficult to recognize that such a lengthy list may be organized into families of systems that share common features. However, the distribution of the variance makes it hard to get a clear picture of the architecture of the family.

This is mainly due to the complexity and interrelations between several types of variance factors.

Identifying Variance Factors

A number of variance factors can be traced in a typical mobile phone manufacturer's product set. The most important ones are listed below. The examples obviously refer to Nokia's product set.

The Handset Category

According to the targeted customer and the price level, mobile handsets (especially those that use a digital protocol for transmission) usually possess remarkably different features.

We usually refer to *low-end,* or *mass* products, intended for a large audience, primarily of young age. Such products are marketed based on aspect and some peculiar features (such as image messaging for GSM-based SMS).

The next category is *high-end,* intended for a specialized audience that is willing to pay more for better services, and accepts complexity of use in exchange.

Finally, *niche products,* such as personal digital assistants (PDA), sit at the top of the price range, and combine the functionality of mobile handsets with that of personal organizers.

Products belonging to different categories share the basic common features, usually those present in the low-end products. However, this is not always true, as, in some cases, specific, peculiar features are present only in low-end products, and not in the others. This is the case, for instance, with the Navi™ universal function key, whose presence affects practically all user interaction features. This makes the categories fuzzy, and renders architecting and design of the relating families problematic.

Communication Standard (Protocol)

Due to the presence of numerous, incompatible wireless communication standards, all mobile handsets should have a variant that supports every protocol, thus enabling them to function (and be sold!) in different places. The transmission frequency is also a variance factor.

The most common standards in digital cellular telephony are GSM 900 (where 900 indicates that the frequency is 900 MHz), GSM 1800, GSM 1900, TDMA, CDMA and PDC. Usually, handset support only one or a few (maximum three) such

standards. Multiple-standard phones are sometimes referred to as *dual band,* with GSM 900/1800 being the most notable.

The variance introduced by the different protocols is more complex than it seems: not only is the way the signal is encoded and decoded different, but the services offered by the network standards (and thus those that have to be supported by the handsets) also vary significantly.

The main example is perhaps SMS messaging (short text messaging), a function that is supported only by GSM networks.

TDMA and CDMA generally have fewer additional services. Japanese PDC, instead, has some peculiar features that pose complex requirements on the handset software. An example is the *direct number* feature, used to redirect data calls. This function has to be performed by the phone, since the PDC network standard, used in Japan, does not distinguish between data and voice calls.

This situation will not ease up with the adoption of the *3rd generation* (3G) networks (that will be based on the WCDMA protocol). On the contrary, the interaction between the existing features and those that are going to be introduced will increase both the complexity of the system-level requirements and the number of services available (e.g. video, multimedia).

User Interface

Usability of mobile handsets is one of the main problems that manufacturers have to face with increasing dedication: the number of offered services is increasing, and the intended audience is growing at a fast pace. The majority of new mobile subscribers belong to the non-expert category, and the market imposes that the handsets allow easy and fast use of the available features.

The Navi™ universal function key, which appears in all Nokia's new low-end products, is an example. It is a soft, context-sensitive key, that allows access to the supposedly most important function, according to the status of the phone. For example, when the phone detects an incoming call and starts ringing, pressing the Navi™ key will answer the call; instead, when the user is writing a text message, it will activate the send message function.

This UI concept is somewhat an extension of Nokia's high-end products solution: two soft function keys, that increase the possibilities of navigation through the menus and double the number of functions that are accessible with a single keystroke, but, on the other hand, increase complexity of use. Such products are intended for users with a higher degree of expertise, who accept having to fiddle with manuals and easily get familiar with multiple function keys.

Along with the soft-keys, several other variance factors (that concern the whole user interface) exist throughout our mobile handset product set. Examples are:

- The size and resolution of the display, that may (and do) vary across the product set. Accordingly, both the type and amount of displayable information and the interaction with the user may change.
- Different ways of activating functions. For instance, the call answering function can be activated by pressing a key on the keyboard, or by pressing the headset button (when the headset is connected), or, in products with a sliding keyboard cover, by flipping the cover open. These possibilities may change or extend when different hardware solutions are adopted.

- The presence of different types of keys. An example is the roller™ key that recently appeared for the first time in Nokia's 7100 series: it enables a mouse-like bi-directional navigation of the display.

With the adoption of wide band networks, and the consequent possibility to transmit higher amounts of data, the user interface will have to support features like video, high-quality audio and multimedia. Thus, these and other variance factors will be adding on, causing an increase in complexity of an already problematic area such as the user interface.

Operating System Services

Mobile handsets rely on a lightweight set of operating system services. They mainly concern hardware resource (power, transmission, memory, etc.) management, as well as manufacturing- and maintenance-related services.

However, the evolution in the user features has generated an increase in complexity of the operating systems. New services, such as audio, video and email, must be handled directly by the OS layer. The introduction of dual-band devices (that operate "intelligently" in two frequencies, according to the quality of the signal and the vicinity of the base-station) is another source of complexity.

Additional requirements will be posed by the introduction of the common, open operating system platforms – in our and several other cases, Symbian's EPOC [Symbianweb]. They will provide new services that will partly replace the existing ones. Sound architectural decisions will have to be made when structuring the future 3G EPOC-based product families that shall constitute a milestone in Nokia's product evolution.

Customer and Country Customization

Nokia now sells products in over a hundred countries. This brings up the need for customization, due both to customer standards and to modifications adopted in different places.

Mobile handsets are sold either through local operators (as is the case in the UK and USA), or directly to the final users.

In the former case (products sold through the operator), and sometimes also in the latter case, some operator-specific customization is required, mainly to integrate the services offered by the handset with those offered by the network. This is the case, for instance, for the SMS-message-based Internet access services, offered through Nokia's Artus platform [Artusweb], available currently only in some products and in selected networks. Another example is security related to the SIM (Subscriber Identity Module) card, which univocally identifies subscribers in GSM networks: some operators pose sophisticated requirements on such processes as SIM card authentication, that have to be supported by phones and networks operating locally.

In either case, the most complex (and, unfortunately, also the most common) customization relates to support for local languages. In particular, supporting ideogram-based languages (such as Chinese and Japanese), Arabic and its variations (where the characters are positional), languages based on the Cyrillic alphabet and Hebrew (just to name a few) requires the user interface (keyboard, display) and memory management to be utterly complex. This is mainly due to the fact that handsets process and store information using the Western transliteration. For

languages where the western transliteration is not univocal, that is, different words transliterate into the same western word, and/or vice versa, this generates obvious problems.

This type of variance spreads across the whole product set (with the exception of products that are not sold in certain "problematic" countries). As long as mobile handset users spread worldwide, managing such issues will be increasingly arduous.

Open Problems

The complexity of the variance factors listed in the previous sections makes it difficult for us to architect, structure, design and model system families. In particular, we feel that future research should concentrate on a number of key areas related to system families.

Requirements Management and Engineering

The lifecycle of products reflects the evolution of the requirements set. Inadequate or poor requirements management and engineering practices inevitably lead to problems in the product, or family, architecture.

Elicitation and capturing are the initial phases. At such early stages of product development, the problem mainly resides in that there is no clear idea of the position of the relating product(s) inside the family.

Consequently, during the requirements modeling and system architecting phase, the similarities and differences with other existing and future products cannot possibly be well defined.

This situation usually leads to problems when the software is implemented:

- At times, work is duplicated, i.e. similar or identical components are implemented separately in different places.
- The architectural patterns are loose: the same system-level requirements may correspond to different design solutions in different products.
- The need for family architecting is recognized only late during product design, when it becomes clear that common features need to be identified and variance factors modeled.

The requirements management and engineering community has so far failed to deliver sufficiently exhaustive answers to the above problems, especially as the telecommunications domain is concerned [Maccari99]. Researchers should put more effort on industrial needs and realities when carrying out their work [Fenton94].

Structuring System Families with Variance

At the moment, the management of product variance within product families is based on distributing requirement specifications for specific products in the hope that the people responsible for the product development projects will notice the commonalties in different products. A more systematic approach is clearly needed. The variance factors and their relationship to the architecture of the family must be made explicit.

Our research group [Karhinen98] has presented an approach to organizing design decisions in an explicit hierarchy, a design decision tree, which can be used to analyze the implications of architectural changes. Perhaps similar hierarchical structures could be used to organize the variance factors pertaining to a product family.

Modeling Product Families

Due to the complexity and spectrum of the variance factors, we have so far experienced difficulties in finding a unique, effective way to model product families.

The ideal modeling technique should have the following characteristics.

- Not cause too big burden for the architects and developers to learn and use.
- Provide an efficient and practical means of communications between the different stakeholders of a system or system family.
- Be based on some standard or well known modeling language (e.g. UML).
- Provide effective means for representing the multidimensional diversity in variance explained in section 2.1.

Recently, a number of European industrial and research organizations have recognized the need to act jointly to address the problems stated in the previous sections (and others that pertain to different domains). The ESAPS project is one of the first large-scale efforts in this direction.

Directions for ESAPS

ESAPS is a large European project that is a part of the Eureka program. The aim of the project is to introduce the system-family approach of software engineering to European industry. The results of the project comprise of processes, methods and tools, and component platforms for system-family engineering. The participants of the project include a number of major European companies that produce software-intensive systems and several research institutions.

The work in ESAPS is divided into four work packages. The first three packages (analysis, processes and methods, derivation of products) provide the techniques, methods, and tools to be validated by the fourth work package in industrial settings through case studies.

We intend to use the results from ESAPS to address some of the problems described in section 2. Therefore, we are currently planning a case study to be conducted in the development of the software for mobile phones. In the following, we outline the study.

A Nokia Case Study: The EPOC-Based System Family

As the Nokia case study for work package 4 of ESAPS, we propose the development of the software for the family of mobile terminals that will run on the new EPOC operating system platform developed by Symbian [Symbianweb]. EPOC represents a major step in mobile terminal technology that will promote modularity and application-oriented development of mobile phones. EPOC will provide a technological base for component-based software development, customizable user interfaces, color support, fit-for-purpose application suites, advanced Internet connectivity, and PC connectivity software.

The reasons for choosing this family are:

- The architectural work on the EPOC family is just starting.

- The development process of the family is in a phase where the study can provide real impact.
- Many new product categories (families) are expected to be developed on EPOC, which brings in new variance factors.

The problem with the EPOC case is that the architectural work should be started before any deliverables can be expected from the three first work packages from ESAPS. However, the previous experiences in software architecture work (e.g. the ARES project [ARES]) will help us to start working with the EPOC family.

For instance the ARES conceptual framework for software architecture (ARES CFSA) [Ran00] gives a model that system developers can follow when designing the architecture of new software. The framework distinguishes the different planes of existence of software (design or write, build, configuration, re-start, and execution) and the component domain of each plane (e.g. modules are write plane components and threads are execution plane components). For each component domain, the architecture is then modeled through four facets: architecturally significant requirements, conceptual model, structures (components and composition), and texture (recurring microstructure). The strength of the CFSA framework appears to be in the comprehensive treatment of the sources of complexity in software-intensive products. However, concrete tools, for instance, modeling languages, are out of the scope of the CFSA.

Research Goals and Approach

As stated above, the goal of the study is to *investigate the effects of adopting the system-family approach* developed in ESAPS. Due to resource considerations and the development status of the EPOC family, we will concentrate on the specification and modeling of the reference architecture for the system-family. A special emphasis will be given on modeling family-level requirements and on the tracing of requirements into the architecture, which is the key problem area. As a formal result, a report will be delivered for the ESAPS-project reporting the findings of the study.

The study will be conducted while the researchers are working as advisors of the system architects of Nokia. This means in-depth exposure to the phenomena concerning the architecture development process. This will also make it possible to see the development process from the system architect's point of view. According to [Orlikowski91], our approach can be characterized as *interpretive* because instead of having a pre-defined set of constructs and instruments for measuring the effects of the system family approach, we will try to describe, interpret, analyze, and understand the studied phenomena from the participant's (the Nokia system architects) point of view. The danger in this kind of research is that it will be reduced into reporting experiences about what works and what does not work without deeper analysis and suggestions for improvement. We must a make a conscious effort to avoid this trap, for instance, by getting the developers at Nokia to be involved in the creation of the final report.

Research Tasks

Because the first three work packages of ESAPS have not yet delivered their results, it is not possible to device a detailed research plan. According to our current understanding, the generic list of research tasks and their ordering is:

1. Identify the common features, variance factors, and family-level requirements in the EPOC-based family.
2. Model the identified common features, variability, and requirements.
3. Design the domain specific reference architecture for the family (and build a formal model).
4. Collect and record the experience data.
5. Analyse and disseminate the results; identify problem for further investigation.

Hopefully, during task 4, early results from ESAPS will be available. Note that collecting of the research data (experiences) must be a continuous process in this kind of study.

Concluding Remarks

The purpose of this paper was to give an overview of the issues pertaining to the software architecture work in the development of the product families of Nokia's mobile handsets. We presented the major categories of variance factors in the domain of mobile handsets and discussed the most important problems in the product family development. Then, we described a case study that attempts to introduce a systematic approach to system family development in Nokia according to the methodologies and techniques developed in ESAPS.

References

[ARES] http://sirio.dit.upm.es/~ares/ and http://hpv17.infosys.tuwien.ac.at/ARES/.

[Artusweb] http://www.nokia.com/networks/17/napf.html.

[ESAPS] Engineering Software Architectures, Processes and Platforms for System Families. ITEA project no 99005, Eureka Σ! 2023 Programme. See http://www.esi.es/esaps/.

[Fenton94] N. Fenton, S. Lawrence Pfleger, R. L. Glass, *Science and substance: a challenge to software engineers,* IEEE Software, July 1994, pp. 86-95.

[Karhinen98] A. Karhinen, J. Kuusela, *Structuring Design Decisions for Evolution*, in proceedings of the Second International ESPRIT ARES Workshop, Spain, LNCS 1429, Springer, 1998, pp. 223-234.

[Kuusela99] J. Kuusela, *Architectural evolution, Nokia Mobile Phones case,* in *Software Architecture,* edited by Patrick Donohoe, Kluwer Academic Publishers, Boston, USA, 1999.

[Maccari99] A. Maccari, *The challenges of requirements engineering in mobile telephones industry,* proceedings of DEXA99, Third International Workshop on Database and Expert Systems Applications, IEEE Computer Society, 1999.

[Nokiaweb] http://www.nokia.com/.

[Orlikowski91] W. Orlikowski, J. J. Baroudi, *Studying information technology in organizations, research approaches and assumptions*, Information Systems Research 2:1, 1991.

[Ran00] Alexander Ran, *ARES conceptual framework for software architecture*. Chapter 1 in M. Jazayeri, A. Ran and F. Linden (eds.), *Software Architecture for Product Families*, Addison-Wesley, 2000.

[Symbianweb] http://www.symbian.com/.

Product Family Methods

Paul Clements

Software engineering Institute, Carnegy Mellon University
Pittsburg, PA 15213, USA
clements@sei.cmu.edu

The Product Family Methods session dealt with how to achieve product family results by solving the practical problems of organization, approach, and process. Organizing for Software Product Lines (Jan Bosch, U. of Karlskrona/Ronneby) exemplified this by describing several different organizational structures observed in actual case studies of organizations using the product family approach. For each organizational scheme presented, its advantages and disadvantages were described, along with conjectures about the size and type of organization in which that scheme would be most effective.

In Comparison of Software Product Family Process Frameworks, Tuomo Vehkomaki and Kari Kansala of Nokia Research Center tried to clear up the confusion among the many process improvement frameworks that exist in the community, and draw conclusions about how each applies to helping initiate, follow, and optimize the process for the product family situation. CMM, SPICE, ISO 9000-3, IEEE 1074, J-STD-016, SE-CMM, IEEE 1220, and EIA/IS-632 were all discussed, and comparisons and contrasts drawn. In the product line realm, the SEI's Product Line Practice Framework, SPC's Synthesis method, Jacobson's Reuse-driven Software Engineering Business approach, and a couple of product line frameworks based on SPICE were all analyzed. The result of the comparison was the proposal for a generic product line process framework, drawing upon the best features of all of the others.

In Issues Concerning Variability in Software Product Lines, Mikael Svahnberg and Jan Bosch (U. Karlskrona/Ronneby) tackled the problem of handling component and other artifact variability in the architecture and supporting infrastructure. Product lines work by using common assets to build a variety of products, and those assets must almost always be tailored (varied) along pre-planned paths in order to serve their purpose in each product. Svahnberg and Bosch lay out several variation mechanisms that are commonly used in product lines. Common mechanisms include inheritance, extensions and extension points, parameterization, using module interconnection languages to vary the system's configuration, and automatic generation of tailored source code.

Finally, in A First Assessment of Development Processes with respect to Product Lines and Component Based Development, Rodrigo Ceron, Juan C. Duenas, and Jun A. De la Puente (Unversidad Politecnica de Madrid) evaluates the Rational Unified Process in light of its ability to support the development of product lines. The evaluation is performed by comparing RUP to known development methods that address part of the product line situation: component development, and reuse-driven software engineering business (and other reuse methods). The authors conclude that what is most needed is more practical experience with the method, although they point out that UML has well-known shortcomings with respect to being able to support the specification of variation points, and that the RUP could be improved with respect to its ability to handle "imported" components provided by outside sources.

F. van der Linden (Ed.): IW-SAPF-3, LNCS 1951, p. 116, 2000.

Organizing for Software Product Lines

Jan Bosch

University of Karlskrona/Ronneby
Department of Software Engineering and Computer Science
SoftCenter, S-372 25, Ronneby, Sweden
Jan.Bosch@ipd.hk-r.se

Abstract

Software product lines have received increasing amounts of attention within the software engineering community, especially from industry. Most authors focus on the technical and process aspects and assume an organizational model consisting of a domain engineering unit and several application engineering units. In our cooperation with several software development organizations applying software product line principles, we have identified several other organizational models that are employed. This article presents a number of organizational models, organized in four main approaches, i.e. development department, business units, domain engineering units and hierarchical domain engineering units. For each approach, its characteristics, applicability and advantages and disadvantages are discussed, as well as an example.

1 Introduction

Achieving reuse of software has been a long standing ambition of the software engineering industry. Every since the paper by [8], the notion of constructing software systems by composing software components has been pursued in various ways. Most proposals to achieving component-based software development assume a market divided into component developers, component users and a market place. However, this proved to be overly ambitious for most types of software. In response, there has been a shift from world-wide reuse of components to organization-wide reuse. Parallel to this development, the importance of an explicit design and representation of the architecture of a software system has become increasingly recognized. The combination of these two insights lead to the definition of software product lines. A software product line consists of a product line architecture, a set of reusable components and a set of products derived from the shared assets.

Existing literature on software product lines tends to focus on the technology and the processes that surround product line based software development. These processes include the design of the software architecture for the product line, the development of the shared software components, the derivation of software products and the evolution of the aforementioned assets. However, generally the organizational structure of software development organizations that is needed for the successful execution of these

F. van der Linden (Ed.): IW-SAPF-3, LNCS 1951, pp. 117–134, 2000.

processes is not discussed. It is, nevertheless, necessary to impose an organization on the individuals that are involved in the product line.

In this article, we discuss a number of organizational models that can be applied when adopting a product line based approach to software development. For each model, we describe in what situations the model is most applicable, the advantages and disadvantages of the model and an example of a company that employs the model. Below, we briefly introduce the models that will be discussed in the remainder of the chapter:

- **Development department**: When all software development is concentrated in a single development department, no organizational specialization exists with either the system family assets or the systems in the family. Instead, the staff at the department is considered to be resource that can be assigned to a variety of projects, including domain engineering projects to develop and evolve the reusable assets that make up the system family.
- **Business units**: The second type of organizational model employs a specialization around the type of systems. Each business unit is responsible for one or a subset of the systems in the family. The business units share the system family assets and evolution of these assets is performed by the unit that needs to incorporate new functionality in one of the assets to fulfil the requirements of the system or systems it is responsible for. On occasion, business units may initiate domain engineering projects to either develop new shared assets or to perform major reorganizations of existing assets.
- **Domain engineering unit**: This model is the suggested organization for software families as presented in the traditional literature, e.g. [5] and [7]. In this model, the domain engineering unit is responsible for the design, development and evolution of the reusable assets, i.e. the software architecture and the components that make up the reusable part of the system family. In addition, business units, often referred to as system or application engineering units, are responsible for developing and evolving the systems built based on the system family assets.
- **Hierarchical domain engineering units**: In cases where an hierarchical product line has been necessary, also a hierarchy of domain units may be required. In this case, often terms such as `platforms' are used to refer to the top-level system family. The domain engineering units that work with specialized product lines use the top-level family assets as a basis to found their own family upon.

Some factors that influence the organizational model, but that we have not mentioned include the physical location of the staff involved in the system family, the project management maturity, the organizational culture and the type of systems. In addition to the size of the system family in terms of the number of systems and system variants and the number of staff members, these factors are important for choosing the optimal model.

The remainder of this paper is organized as follows. In section 2 until 5, the four aforementioned organizational models are discussed in more detail. Section 6 discusses the aforementioned factors and their effects on selecting the optimal organizational model. Finally, related work is discussed in section 7 and the article is concluded in section 8.

2 Development Department

The development department model imposes no permanent organizational structure on the architects and engineers that are involved in the software product line. All staff members can, in principle, be assigned to work with any type of asset within the family. Typically, work is organized in projects that dynamically organize staff members in temporary networks. These projects can be categorized into domain engineering projects and application (or system) engineering projects. In the former, the goal of the project is the development of a new reusable asset or a new version of it, e.g. a software component. The goal is explicitly not a system or product that can be delivered to internal or external customers of the development department. The system engineering projects are concerned with developing a system, either a new or a new version, that can be delivered to a customer. Occasionally, extensions to the reusable assets are required to fulfil the system requirements that are more generally applicable than just the system under development. In that case, the result of the system engineering project may be a new version of one or more of the reusable assets, in addition to the deliverable system.

In figure 1, the development department model is presented graphically. Both the reusable product line assets and the concrete systems built based on these assets are developed and maintained by a single organizational unit.

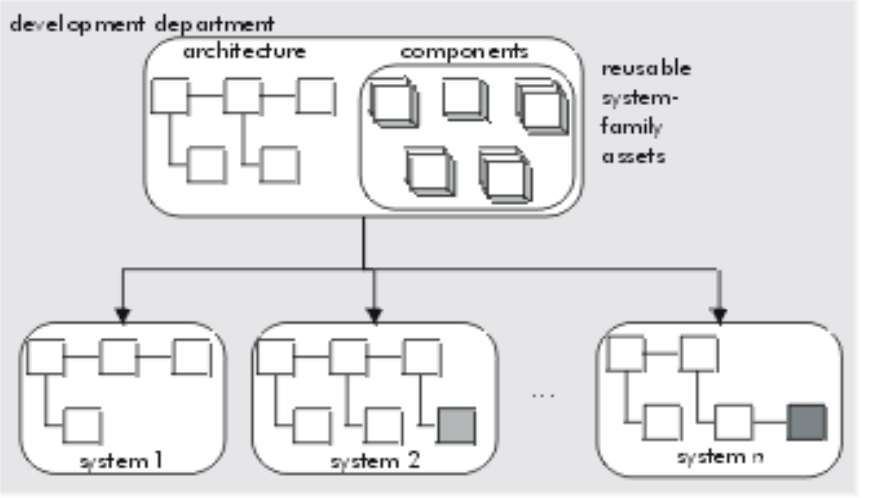


Figure 1. Development department model

2.1 Applicability

The development department model is primarily applicable for relatively small organizations and for consultancy organizations, i.e. organizations that sell projects rather than products to their customers. Based on our experience, our impression is that this model works up to around 30 software-related staff members in product-based organizations. If the number of staff members exceeds 30, generally some kind of organizational restructuring is required anyhow, independent of the use of a product line.

2.2 Advantages and disadvantages

The development department model has, as most things in life, a number of advantages and disadvantages. The primary advantage is simplicity and ease of communication. Since all staff members are working within the same organizational context, come in contact with all parts of the system family and have contact with the customers, the product line can be developed and evolved in a very efficient manner with little organizational and administrative overhead. A second advantage is that, assuming that a positive attitude towards reuse-based software development exists within the department, it is possible to adopt a software product line approach without changing the existing organization, which may simplify the adoption process.

The primary disadvantage of this approach is that it is not scalable. When the organization expands and reaches, e.g., around 30 staff members, it is necessary to reorganize and to create specialized units. A second disadvantage is that typically within organizations, staff members are, depending on the local culture, more interested in either domain engineering or system engineering, i.e. it has higher status in the informal organization to work with a particular type of engineering. The danger is that the lower status type of engineering is not performed appropriately. This may lead to highly general and flexible reusable components, but systems that do not fulfil the required quality levels, or visa versa.

2.3 Example

A company that employs this organizational model is Securitas Larm, Sweden. All their product development, i.e. hardware and software, is concentrated in a single development department. This department maintains a product line in the domain of fire-alarm systems, as described in [2]. The department has an engineering staff about 25 persons, so it fits our applicability requirement. In fact, up to a number of years ago, development was organized in product business units. Each product unit was responsible for sales, marketing, installation and development of the product. However, especially development did not function well in this organizational form. Generally only up to five engineers worked with the product development, which was to few to create an effective development organization. Consequently, Securitas Larm decided to reorganize development into a single development department.

3 Business units

The second organizational model that we discuss is organized around business units. Each business unit is responsible for the development and evolution of one or a few products in the software product line. The reusable assets in the product line are shared by the business units. The evolution of shared assets is generally performed in a distributed manner, i.e. each business unit can extend the functionality in the shared assets, test it and make the newer version available to the other business units. The initial development of shared assets is generally performed through domain engineering projects. The project team consists of members from all or most business units. Generally, the business units most interested in the creation of, e.g. a new software component, put the largest amount of effort in the domain engineering project, but all business units share, in principle, the responsibility for all common assets.

Depending on the number and size of the business units and the ratio of shared versus system specific functionality in each system, we have identified three levels of maturity, especially with respect to the evolution of the shared assets:

Unconstrained model. In the unconstrained model, any business unit can extend the functionality of any shared component and make it available as a new version in the shared asset base. The business unit that performed the extension is also responsible for verifying that, where relevant, all existing functionality is untouched and that the new functionality performs according to specification.

A typical problem that companies using this model suffer from is that, especially software components, are extended with too system-specific functionality. Either the functionality has not been generalized sufficiently or the functionality should have been implemented as system-specific code, but for internal reasons, e.g. implementation efficiency or system performance, the business unit decided to implement the functionality as part of the shared component.

These problems normally lead to the erosion or degradation of the component, i.e. it becomes, over time, harder and less cost-effective to use the shared component, rather than developing a system-specific version of the functionality. As we discussed in [2], some companies have performed component reengineering projects in which a team consisting of members from the business units using the component, reengineers the component and improves its quality attributes to acceptable levels. Failure to reengineer when necessary may lead to the situation where the product line exists on paper, but where the business units develop and maintain system-specific versions of all or most components in the product line, which invalidates all advantages of a software product line approach, while maintaining some of the disadvantages.

Asset responsibles. Especially when the problems discussed above manifest themselves in increasing frequency and severity, the first step to address these problems is to introduce asset responsibles. An asset responsible has the obligation to verify that the evolution of the asset is performed according to the best interest of the organization as a whole, rather than optimal from the perspective of a single business unit. The asset responsible is explicitly not responsible for the implementation of new requirements. This task is still performed by the business unit that requires the additional functionality. However, all evolution should occur with the asset responsible's consent and

before the new version of the asset is made generally accessible, the asset responsible will verify through regression testing and other means that the other business units are at least not negatively affected by the evolution. Preferably, new requirements are implemented in such a fashion that even other business units can benefit from them. The asset responsible is often selected from the business unit that makes most extensive and advanced use of the component.

Although the asset responsible model, in theory at least, should avoid the problems associated with the unconstrained model, in practice it often remains hard for the asset responsible to control the evolution. One reason is that time-to-market requirements for business units often are prioritized by higher management, which may force the asset responsible to accept extensions and changes that do not fulfil the goals, e.g. too system-specific. A second reason is that, since the asset responsible does not perform the evolution him or herself, it is not always trivial to verify that the new requirements were implemented as agreed upon with the business unit. The result of this is that components still erode over time, although generally at a lower pace than with the unconstrained model.

Mixed responsibility. Often, with increasing size of the system family, number of staff and business units, some point is reached where the organization still is unwilling to adopt the next model, i.e. domain engineering units, but wants to assign the responsibility for performing the evolution assets to a particular unit. In that case, the mixed responsibility model may be applied. In this model, each business unit is assigned the responsibility for one or more assets, in addition to the system(s) the unit is responsible for. The responsibility for a particular asset is generally assigned to the business unit that makes the most extensive and advanced use of the component. Consequently, most requests for changes and extensions will originate from within the business unit, which simplifies the management of asset evolution. The other business units have, in this model, no longer the authority to implement changes in the shared component. Instead, they need to issue requests to the business unit responsible for the component whenever an extension or change is required.

The main advantage of this approach is the increased control over the evolution process. However, two potential disadvantages exist. First, since the responsibility for implementing changes in the shared asset is not always located at the business unit that needs those changes, there are bound to be delays in the development of systems that could have been avoided in the approaches described earlier. Second, each business unit has to divide its efforts between developing the next version of their system and of the component(s) it is responsible for. Especially when other business units have change requests, these may conflict with the ongoing activities within the business unit and the unit may prioritize its own goals over the goals of other business units. In addition, the business unit may extend the components it is responsible for in ways that are optimized for its own purposes, rather than for the organization as a whole. These developments may lead to conflicts between the business units and, in the worst case, the abolishment of the product line approach.

Conflicts. The way the software product line came into existence is, in our experience, an important factor in the success or failure of a family. If the business units already exist and develop their systems independently and, at some point, the software product

line approach is adopted because of management decisions, conflicts between the business units are rather likely because giving up freedom that one had up to that point in time is generally hard. If the business units exist, but the product line gradually evolves because of bottom-up, informal cooperation between staff in different business units, this is an excellent ground to build a product line upon. However, the danger exist that when cooperation is changed from optional to obligatory, tensions and conflicts appear anyhow. Finally, in some companies, business units appear through an organic growth of the company. When expanding the set of systems developed and maintained by the company, at some point, a reorganization into business units is necessary. However, since the staff in those units earlier worked together and used the same assets, both the product line and cooperation over business units develop naturally and this culture often remains present long after the reorganization, especially when it is nurtured by management. Finally, conflicts and tensions between business units must resolved by management early and proactively since they imply considerable risk for the success of the product line.

In figure 2, the business unit model is presented graphically. The reusable system-family assets are shared by the business units, both with respect to use as well as to evolution.

3.1 Applicability

As discussed in section 2, when the number of staff members is too low, e.g. below 30, the organization in business units is often not optimal since too few people are working together and the communication overhead over unit boundaries is too large. On the other hand, our hypothesis, based on a number of cases that we have studied, is that when the number of staff members exceeds 100, domain engineering units may become necessary to reduce the n-to-n communication between all business units to a one-to-n communication between the domain engineering unit and the system engineering units. Thus, with respect to staff size, we believe that the optimal range for the business unit model is between 30 and 100, although this, to a large extent, depends on the specific context as well.

3.2 Advantages and disadvantages

The advantage of this model is that it allows for effective sharing of assets, i.e. software architectures and components, between a number of organizational units. The sharing is effective in terms of access to the assets, but in particular the evolution of assets (especially true for the unconstrained and the asset responsible approaches). In addition, the approach scales considerably better than the development department model, e.g. up to 100 engineers in the general case.

The main disadvantage is that, due to the natural focus of the business units on systems (or products), there is no entity or explicit incentive to focus on the shared assets. This is the underlying cause for the erosion of the architecture and components in the system family. The timely and reliable evolution of the shared assets relies on the

organizational culture and the commitment and responsibility felt by the individuals working with the assets.

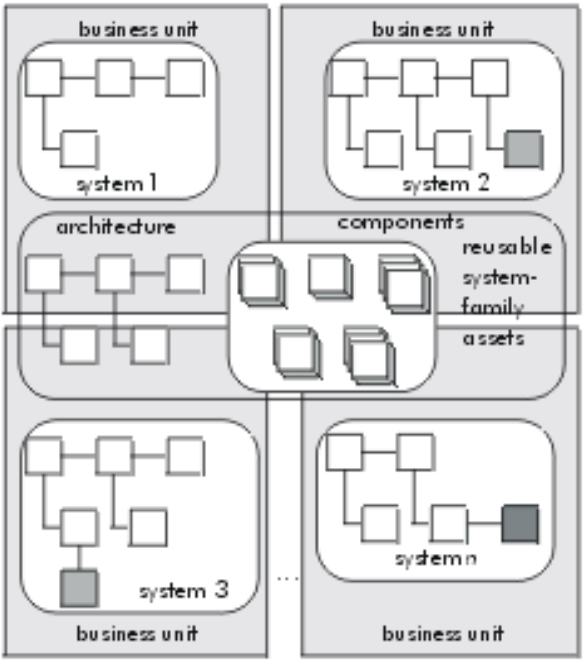


Figure 2. Business unit model

3.3 Example

Axis Communications, Sweden, employs the business unit model. Their storage-server, scanner-server and camera-server products are developed by three business units. These business units share a common product line architecture and a set of more than ten object-oriented frameworks that may be extended with system-specific code where needed. Initially, Axis used the unconstrained model with relatively informal asset responsibles, but recently the role of asset responsibles has been formalized and they now have the right to refuse new versions of assets that do not fulfil generality, quality and compatibility requirements. The assets responsibles are taken from the business units that make the most extensive and advanced use of the associated assets. Within the organization, discussions are ongoing whether an independent domain engineering unit, alternatively, a mixed responsibility approach are needed to guarantee the proper evolution of assets. Whenever new assets or a major redesign of some existing asset is needed, Axis has used domain engineering projects, but `disguised' these projects as system engineering projects by developing prototype systems. The advantage of the latter is that the integration of the new asset with the existing assets is automatically verified as part of the domain engineering project.

4 Domain Engineering Unit

The third organizational model for software product lines is concerned with separating the development and evolution of shared assets from the development of concrete systems. The former is performed by a, so-called, domain engineering unit, whereas the latter is performed by system engineering units. System engineering units are sometimes referred to as application engineering units.

The domain engineering unit model is typically applicable for larger organizations, but requires considerable amounts of communication between the system engineering units, that are in frequent contact with the customers of their systems, and the domain engineering unit that has no direct contact with customers, but needs a good understanding of the requirements that the system engineering units have. Thus, one can identify flows in two directions, i.e. the requirements flow from the system engineering units towards the domain engineering unit and the new versions of assets, i.e. the software architecture and the components of system family, are distributed by the domain engineering unit to the system engineering units.

The domain engineering unit model exists in two alternatives, i.e. an approach where only a single domain engineering unit exists and, secondly, an approach where multiple domain engineering units exist. In the first case, the responsibility for the development and evolution of all shared assets, that software architecture and the components, is assigned to a single organizational unit. This unit is the sole contact point for the system engineering units that construct their systems based on the shared assets.

The second alternative employs multiple domain engineering units, i.e. one unit responsible for the design and evolution of the software architecture for the product line and, for each architectural component (or set of related components), a component engineering unit that manages the design and evolution of the components. Finally, the system engineering units are, also in this alternative, concerned with the development of systems based on the assets. The main difference between the first and second alternative is that in the latter, the level of specialization is even higher and that system engineering units need to interact with multiple domain engineering units.

In figure 3, the organizational model for using domain engineering unit is presented. The domain engineering unit is responsible for the software architecture and components of the product line, whereas the system engineering units are responsible for developing the systems based on the shared assets.

4.1 Applicability

Especially smaller companies are very sceptical of domain engineering units. One of the concerns is that, just because domain engineering units are concerned with reusable assets, rather than systems that are relevant for customers, these units may not be as focused on generating added value, but rather lose themselves in aesthetic, generic, but useless abstractions. However, based on our experience, our impression is that when the number of staff members working within a system family exceeds around 100 software engineers, the amount of overhead in the communication between the business

units causes a need for an organizational unit or units specialized on domain engineering.

Multiple rather than a single domain engineering unit become necessary when the size of the domain engineering unit becomes too large, e.g. more than 30 software engineers. In that case, it becomes necessary to create multiple groups that focus on different component sets within system family software architecture. In some cases, although component engineering units exist, no explicit system family software architecture unit is present. Rather, a small team of software architects from the component engineering units assumes the responsibility for the overall architecture.

Finally, at which point the complexities of software development even exceed the domain engineering unit approach is not obvious, but when the number of software engineers is in the hundreds the hierarchical domain engineering units model, discussed in the next section, may become feasible.

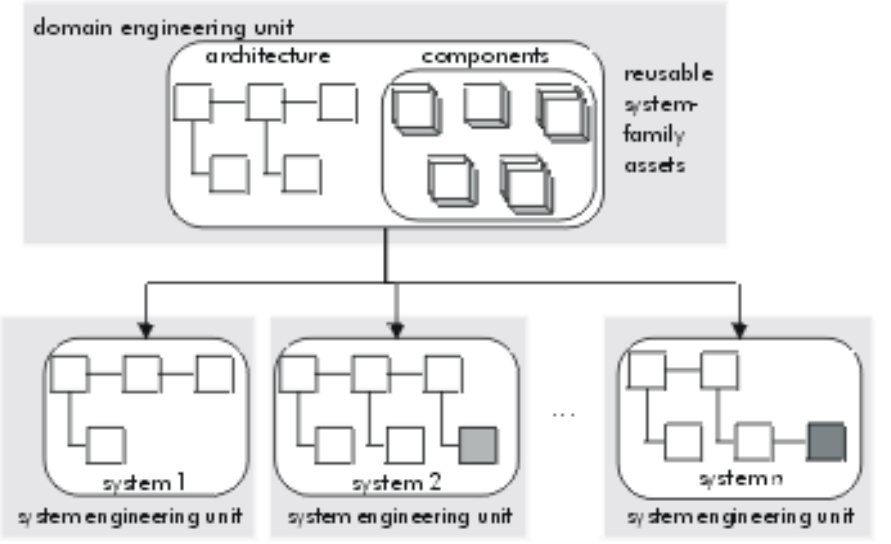


Figure 3. The domain engineering unit model

4.2 Advantages and disadvantages

Despite the scepticism in, especially smaller organizations, the domain engineering unit model has a number of important advantages. First, as mentioned, it removes the need for n-to-n communication between the business units, and reduces it to one-to-n communication. Second, whereas business units may extend components with too system-specific extensions, the domain engineering unit is responsible for evolving the components such that the requirements of all systems in the product line are satisfied. In addition, conflicts can be resolved in a more objective and compromise-oriented fashion. Finally, the domain engineering unit approach scales up to much larger numbers of software engineering staff than the aforementioned approaches.

Obviously, the model has some associated disadvantages as well. The foremost is the difficulty of managing the requirements flow towards the domain engineering unit, the balancing of conflicting requirements from different system engineering units and the subsequent implementation of the selected requirements in the next version of the assets. This causes delays in the implementation of new features in the shared assets, which, in turn, delays the time-to-market of systems. This may be a major disadvantage of the domain engineering unit model since time-to-market is the primary goal of many software development organizations. To address this, the organization may allow system engineering units to, at least temporarily, create their own versions of shared assets by extending the existing version with system-specific features. This allows the system engineering unit to improve its time-to-market while it does not expose the other system engineering units to immature and instable components. The intention is generally to incorporate the system-specific extensions, in a generalized form, into the next shared version of the component.

4.3 Example

The domain engineering unit model is used by Symbian. The EPOC operating system consists of a set of components and the responsibility of a number of subsets is assigned to specialized organizational units. For each device family requirement definition (DFRD), a unit exists that composes and integrates versions of these components into a release of the complete EPOC operating system to the partners of Symbian. The release contains specific versions and instantiations of the various components for the particular DFRD. Some components are only included in one or a few of the DFRDs.

5 Hierarchical Domain Engineering Units

As we discussed in the previous section, there is an upper boundary on the size of an effective domain engineering unit model. However, generally even before the maximum staff member size is reached, often already for technical reasons, an additional level has been introduced in the software product line. This additional layer contains one or more specialized product lines that, depending on their size and complexity can either be managed using the business unit model or may actually require a domain engineering unit.

In the case that a specialized product line requires a domain engineering unit, we have, in fact, instantiated the hierarchical domain engineering units model that is the topic of this section. This model is only suitable for a large or very large organization that has an extensive family of products. If, during the design or evolution of the product line, it becomes necessary to organize the product line in a hierarchical manner and a considerable number of staff members is involved in the product line, then it may be necessary to create specialized domain engineering units that develop and evolve the reusable assets for a subset of the systems in the family.

The reusable product line assets at the top level are frequently referred to as a platform and not necessarily identified as part of the product line. We believe, however, that it is relevant to explicitly identify and benefit from the hierarchical nature of these assets. Traditionally, platforms are considered as means to provide shared functionality, but without imposing any architectural constraints. In practice, however, a platform does impose constraints and when considering the platform as the top-level product line asset set, this is made more explicit and the designers of specialized product lines and family members will perform derive the software architecture rather than design it.

In figure 4, the hierarchical domain engineering units model is presented graphically. For a subset of the systems in the product line, a domain engineering unit is present that develops and maintains the specialized product line software architecture and the associated components. Only the components specific for the subset in the product line are the responsibility of the specialized domain engineering unit. All other components are inherited from the overall product line asset base. The specialized domain engineering unit is also responsible for integrating the specialized with the general reusable assets.

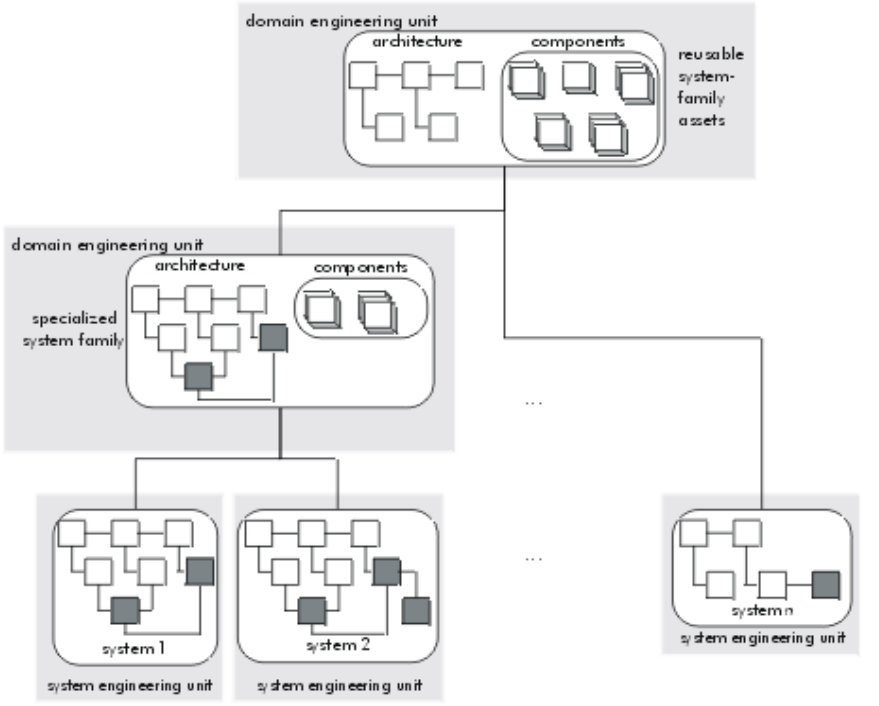


Figure 4. Hierarchical domain engineering unit model

5.1 Applicability

As mentioned in the introduction, the hierarchical domain units model becomes the preferred model when the number and variability of systems in the family is large or very large and considerable numbers of staff members, i.e. hundreds, are involved. Consequently, the model is primarily suitable in large organizations and long-lived systems in the family, since the effort and expenses involved in building up this organizational model are substantial.

The complexities involved in the implementation and use of this organizational model are beyond the scope of this article, but a considerable maturity with respect to software development projects is required for this approach to succeed. This model is the fourth and most complex model that we discuss and if the product line cannot be captured within this model, it is reasonable to assume that the scope of the family has been set too wide.

5.2 Advantages and disadvantages

The advantages of this model include its ability to encompass large, complex product lines and organize large numbers of engineers. None of the organizational models discussed earlier scales up to the hundreds of software engineers that can be organized using this model.

The disadvantages include the considerable overhead that the approach implies and the difficulty of achieving agile reactions to changed market requirements. For instance, in the case of a system part of a specialized product line that, in its turn, is part of the top level product line, a change in a top level component required for addressing a market opportunity or threat may take considerable amounts of synchronization effort. When this occurs, a delicate balance needs to be found between allowing system engineering units to act independent, including the temporary creation of system-specific versions of product line components, versus capitalizing on the commonalities between products and requiring system engineering units to use shared versions of components.

5.3 Example

Although we are aware of companies applying this model as part of their product line development, we currently have no permission to mention any.

6 Influencing Factors

Up to this point, we have presented the size of the product line and the engineering staff involved in the development and evolution of the product line as the primary factors in selecting the appropriate organizational model. Although, in our experience, the above factors indeed are the most prominent, several factors exist that should be

allowed to influence the selection decision as well. Below, we present some factors that we have identified in industry as relevant in this context.

6.1 Geographical distribution

Despite the emergence of a variety of technological solutions aiming at reducing the effects of geographical location, e.g. telephone, e-mail, video conferencing and distributed document management, the physical location of the staff involved in the software product line still plays a role. It simply is more difficult to maintain effective and efficient communication channels between teams that are in disparate locations and, perhaps even, time zones, than between teams that are local to each other. Therefore, units that need to exchange much information should preferably be located closer to each other than units that can cooperate with less information.

For instance, geographical distribution of the teams developing the systems in the family may cause a company to select the domain engineering unit model because it focuses the communication between the domain engineering unit and each system engineering unit, rather than the n-to-n communication required when using the business unit model.

6.2 Project management maturity

The complexity of managing projects grows exponentially with the size of the project (in virtually any measure). Therefore, the introduction of a software product line approach requires, independent of the organizational model, a relatively high level of maturity with respect to project management. Projects need to be synchronized over organizational boundaries and activities in different projects may be depending on each other, which requires experience and pro-activeness in project management.

To give an example, incorporating new functionality in a product line component at Axis Communications requires communication with the other business units at the start, the actual execution and at the end of the project. At the start because it should be verified that no other business unit is currently including the same or related functionality. During the project, to verify that the included functionality and the way in which it is implemented are sufficiently general and provide as much benefit as possible to the other business units. After the end of the project, to verify that the new version of the component provides backward compatibility to systems developed by the other business units.

6.3 Organizational culture

The culture of an organization is often considered to be hard to use concept, which is obviously the case. However, the attitude that each engineer has towards the tasks that he or she is assigned to do and the value patterns exhibited by the informal organizational groups have a major influence on the final outcome of any project. Thus, if a kind of `cowboy' or `hero' culture exists in which individual achievements are valued

higher than group achievements, then this attitude can prove to be a serious inhibitor of a successful software product line approach that is highly dependent on a team culture that supports interdependency, trust and compromise.

For instance, at one company, which will remain unnamed, we discussed the introduction of a software product line approach. The company had extensive experience in the use of object-oriented frameworks and within each business unit reuse was widespread and accepted. However, when top management tried to implement product line based reuse, business unit managers revolted and the initiative was cancelled. The reason, it turned out, was that each business unit would have to sacrifice its lead architect(s) for a considerable amount of time during the development of the reusable product line assets. In addition, the conversion would delay several ongoing and planned projects. These two effects of adopting a product line approach would, among others, lead to highly negative effects on the bonuses received by, especially, business unit management. One explanation could be that these managers were selfish people that did not consider what was best for the company as a whole. However, our explanation is that top management had, under many years, created a culture in which business units were highly independent profit centres. This culture conflicted directly with the product line approach top management tried to introduce.

6.4 Type of systems

Finally, an important factor influencing the optimal organizational model, but also the scope and nature of the system family, is the type of systems that make up the family. Systems whose requirements change frequently and drastically, e.g. due to new technological possibilities, are substantially less suitable for large up-front investments that a wide scoped, hierarchical software product line approach may require, than systems with relatively stable requirement sets and long lifetimes. Medical and telecommunication (server-side) systems are typical systems that have reasonably well understood functionality and that need to be maintained for at least a decade and often considerably longer.

For instance, we earlier discussed the possibility for consultancy companies that typically are project based to adopt a software product line approach. Since subsequent projects often are in the same domain, the availability of a product line architecture and a set of reusable components may substantially reduce lead time and development cost. However, the investment taken by such a company to develop these assets can never be in the same order of magnitude as a product-based company with clear market predictions for new products. The consultancy company has a significantly higher risk that future projects are not in exactly the same domain, but an adjacent, invalidating or at least reducing the usefulness of the developed assets. Consequently, investment and risk always need to be balanced appropriately.

7 Related Work

As we discussed in the introduction, most publications in the domain of software product lines address issues different from the organizational ones. Macala et al. [7] and Dikel et al. [5] were among the first publications that describe experiences from using software product lines in an industrial context. Although the authors do address organizational, management and staffing issues, both assume the domain engineering unit model and present it as the de-facto organizational model. Jacobsen et al. [6] also discuss organizational issues, but focus on a number of roles that should be present and do not address the overall organization of software product line based development. In [4], the authors address organizational issues of software product line. The authors identify four functional groups, i.e. the architecture group, the component engineering group, the product line support group and the product development group. The authors identify that these functional groups may be mapped to organizational units in various ways. Finally, Bayer et al. [1] discuss a methodology for developing software product lines and discuss organizational guidelines, but no organizational models.

8 Conclusion

In this article, we have discussed four organizational models for software product lines and discussed, based on our experiences, the applicability of the model, the advantages and disadvantages and an example of an organization that employs the particular model. Below, the four models are briefly summarized:

- **Development department**: In this model software development is concentrated in a single development department, no organizational specialization exists with either the software product line assets or the systems in the family. The model is especially suitable for smaller organizations. We have seen successful instances of this model up to 30 software engineers. The primary advantages are that it is simple and communication between staff members is easy, whereas the disadvantage is that the model does not scale to larger organizations.
- **Business units**: The second type of organizational model employs a specialization around the type of systems in the form of business units. The business units share the product line assets and evolution of these assets is performed by the unit that needs to incorporate new functionality in one of the assets to fulfil the requirements of the system or systems it is responsible for. Three alternatives exist, i.e. the unconstrained model, the asset responsibles model and the mixed responsibility model. The model is often used as the next model in growing organizations once the limits of the development department model are reached. Some of our industrial partners have successfully applied this model up to 100 software engineers. An advantage of the model is that it allows for effective sharing of assets between a set of organizational units. A disadvantage

is that business units easily focus on the concrete systems rather than on the reusable assets.

- **Domain engineering unit**: In this model, the domain engineering unit is responsible for the design, development and evolution of the reusable assets, i.e. the software architecture and the components that are make up the reusable part of the software product line. In addition, system engineering units are responsible for developing and evolving the systems built based on the product line assets. The two alternatives include the single domain engineering unit model and the multiple domain engineering units model. In the latter case, one unit is responsible for the product line architecture and others for the reusable software components. The model is widely scalable, from the boundaries where the business unit model reduces effectiveness up to several hundreds of software engineers. One advantage of this model is that it reduces communication from n-to-n in the business unit model to one-to-n between the domain engineering unit and the system engineering units. Second, the domain engineering unit focuses on developing general, reusable assets which addresses one of the problems with the aforementioned model, i.e. too little focus on the reusable assets. One disadvantage is the difficulty of managing the requirements flow and the evolution of reusable assets in response to these new requirements. Since the domain engineering unit needs to balance the requirements of all system engineering units, this may negatively affect time-to-market for individual system engineering units.
- **Hierarchical domain engineering units**: In cases where an hierarchical product line has been necessary, also a hierarchy of domain units may be required. The domain engineering units that work with specialized product lines use the top-level assets as a basis to found their own product line upon. This model is applicable especially in large or very large organizations with a large variety of long-lived systems. The advantage of this model is that it provides an organizational model for effectively organizing large numbers of software engineers. One disadvantage is the administrative overhead that easily builds up, reducing the agility of the organization as a whole, which may affect competitiveness negatively.

Finally, we have discussed a number of factors that influence the organizational model that is optimal in a particular situation. These factors include geographical distribution, project management maturity, organizational culture and the type of systems.

References

[1] J. Bayer, O. Flege, P. Knauber, R. Laqua, D. Muthig, K. Schmid, T. Widen, J.M. DeBaud, 'PuLSE: A Methodology to Develop Software Product Lines, *Symposium on Software Reuse*, 1999.

[2] Jan Bosch, 'Product-Line Architectures in Industry: A Case Study', *Pro-*

ceedings of the 21st International Conference on Software Engineering pp. 544-554, May 1999.

[3] Jan Bosch, *Design and Use of Software Architectures: Adopting and Evolving a Product Line Approach*, Addison Wesley Longman (forthcoming), ISBN 0-201-67494-7, May 2000.

[4] P. Clements, L. Northrop, 'A Framework for Software Product Line Practice - Version 1.0', *Software Engineering Institute*, Carnegie Mellon, September 1998.

[5] D. Dikel, D. Kane, S. Ornburn, W. Loftus, J. Wilson, 'Applying Software Product-Line Architecture,' *IEEE Computer*, pp. 49-55, August 1997.

[6] I. Jacobsen, M. Griss, P. Jönsson, *Software Reuse - Architecture, Process and Organization for Business Success*, Addison-Wesley, 1997.

[7] R.R. Macala, L.D. Stuckey, D.C. Gross, 'Managing Domain-Specific Product-Line Development,' *IEEE Software*, pp. 57-67, 1996.

[8] M. D. McIlroy, 'Mass Produced Software Components,' in '*Software Engineering*,' Report on A Conference Sponsored by the NATO Science Committee, P. Naur, B. Randell (eds.), Garmisch, Germany, 7th to 11th October, 1968, NATO Science Committee, 1969.

A Comparison of Software Product Family Process Frameworks

Tuomo Vehkomäki, Kari Känsälä

Nokia Research Center
Helsinki, Finland
tuomo.vehkomaki@nokia.com, kari.kansala@nokia.com

Abstract: A number of product family process frameworks has been published recently. These frameworks focus on different aspects of product family based development. We have investigated a variety of publicly available product family frameworks and chosen four of the variants for maximum coverage of different viewpoints. We first propose a reference product line process framework. With the help of the reference framework, the chosen source frameworks are correlated and compared at the level of individual activities. Both in the reference framework and in the comparison, we stress domain engineering as one of the most essential activities.

Introduction

The objective of this study is to create a generic software product line process framework that can be used as a reference model to compare product line approaches known by today's industry. The objective has not been to create another process framework, but a benchmark of existing frameworks. The generic framework is best used to organize references to the actual information sources, such as the compared product family process frameworks, or proprietary process descriptions within a specific industry.

The product line approaches of interest to us represent rather different viewpoints. Therefore the generic framework needs to be comprehensive enough to allow mapping between product line approaches with different coverage.

In our terminology, the Generic Product Line Process (GPLP) covers the actual software development cycle for all levels of granularity: systems, products, platforms, and components. The term Generic Product Line Process Framework (GPLPF) includes GPLP plus supporting process categories i.e. the transition to product line, product portfolio management, and third party product acquisition and subcontracting.

The section 2 of this text introduces the source frameworks that contribute to the generic product line process framework and the comparison. Section 3

F. van der Linden (Ed.): IW-SAPF-3, LNCS 1951, pp. 135–145, 2000.

describes the generic product line process framework and section 4 compares the source frameworks with the proposed reference framework.

Source Frameworks

We have initially investigated traditional software and systems engineering frameworks. With emergence of frameworks that explicitly deal with product lines, we have included those frameworks in our comparison.

Using SPICE v2.0 [SPICE96] as a skeleton, an extensive comparison of existing software and systems engineering frameworks was presented in 1997 [Nyström97]. The compared frameworks are listed in Table 1. Based on the comparison and existing software processes in Nokia Business Units, a customized version of SPICE v2.0 called NRC Software Process Framework was developed at Nokia Research Center [Känsälä99]. A typical model of industrial product process categories based on SPICE is illustrated in Figure 1.

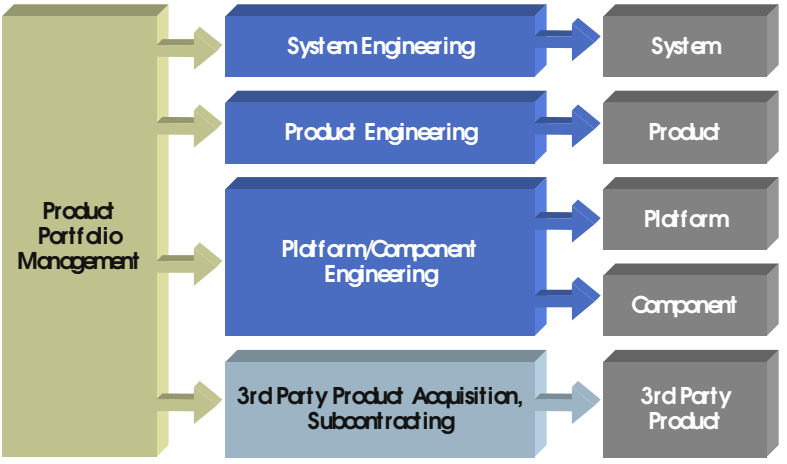


Fig. 1. Typical product process framework. The software processes support development of systems that are composed of four layers: system, product, platform, and component. The depicted framework does not yet include product-line specific activities.

Table 1. Software and systems engineering models compared by Nystöm [Nyström97].

PF	Full name	Status / Version	Released
SPICE	Software Process Improvement and Capability dEtermination	V2.0	1996
CMM	SW Capability Maturity Model/ SEI	V1.1	1993
ISO 9000-3	Guidelines for the Application of ISO 9001 to the Design, Development, Supply, Installation and Maintenance of Computer Software	Draft International Standard	1996

ISO 12207	Software Life Cycle Processes		1995
IEEE 1074	Standard for Developing Software Life Cycle Processes		1995
J-STD-016	Standard for Information Technology, Software Life Cycle Processes, Software Development and Acquirer-Supplier Agreement	Interim Standard	1995
SE-CMM	Systems Engineering Capability Maturity Model/ SEI	V1.1	1995
IEEE 1220	Standard for Application and Management of the Systems Engineering process	Trial-use	1994
EIA/IS-632	Systems Engineering	Interim Standard	1994

The 1997 comparison is used as a background for the comparison of software product line process related models. Starting in the early 90's and more frequently since 1997, several frameworks related to software product lines have been published. A representative set of product line frameworks is listed in Table 2 and summarized in the rest of this section. The listed frameworks are included in the comparison of Section 4.

Table 2. Software product line process frameworks of our comparison.

Framework	Full name [reference]	Status / Version	Released
GPLP	Generic Product Line Process (see section 3.)	Initial	Mar-00
SEI FSPLP	Software Engineering Institute: Framework for Software Product Line Practice / [Clements99]	V2.0	Jul-99
Synthesis, DsE	Domain-specific Engineering (DsE) [Campbell99] based on Synthesis [RSP93]	Presented in Reuse'99	Apr-99
RSEB	Reuse-driven SW Engineering Business [Jacobson97]	Book / ACM Press	Jun-97
SPICE, NRC SPF	Nokia Research Center Software Process Framework [Känsälä99] based on SPICE v2.0 [SPICE96]	V1.1	May-98

SEI Framework for Software Product Line Practice

The approach used by SEI is to identify foundational concepts underlying software product lines and activities to be considered when creating a product line [Clements99]. The listed practice areas comprise an extensive set of competencies and issues necessary to consider for successful adoption of product line based reuse. The viewpoint supports product line planning and management, rather than gives concrete instructions on implementing specific engineering tasks.

Synthesis and Domain-specific Engineering

Synthesis [RSP93] by Software Productivity Consortium is an extensive description of processes related to domain engineering. Synthesis also includes creation of process support for the application engineering. Synthesis does not explicitly address transition to product line based reuse but describes two process variants for different levels of organizational reuse capability.

Domain-specific Engineering (DsE) continues from the basis of Synthesis and relies on parallel domain engineering and application engineering activities in the traditional way of domain engineering. In addition to plain domain engineering, Domain-specific Engineering has explicit activities of domain management, process engineering, and project support [Campbell99].

Reuse-driven Software Engineering Business

Reuse-driven Software Engineering Business (RSEB) [Jacobson1997] describes a systematic model for implementing reuse. The description is tightly coupled with object-oriented analysis and design, the Unified Modeling Language (UML) and layered software architecture. Instructions on how to do analysis, design, implementation, and validation are given and less effort is put on management issues. For a mature organization, the approach may be used as a guide to implement reuse.

The actual process is a derivative of the traditional Domain Engineering/Application Engineering split and has separate activities for

- Application Family Engineering
- Component System Engineering
- Application System Engineering.

Application Family Engineering and Component System Engineering can be considered two separate variations of accustomary domain engineering. Application Family Engineering works at a high level of abstraction to develop a conceptual model and a common layered architecture for all product line members. Component System Engineering works at lower level of abstraction to develop functional building blocks for the layered product platform.

In addition to engineering activities, RSEB also includes an explicit set of activities that support the transition to reuse.

NRC Software Process Framework

Being based on SPICE, NRC Software Process Framework [Känsälä99] is a traditional software engineering process framework which does not cover the product family dimension i.e. it deals with single systems only. The framework consists of 29 processes partitioned to five categories:

- Customer-supplier process category
- Engineering process category
- Support process category
- Management process category
- Organization process category

Being comprehensive also beyond engineering activities, it complements the product line approaches presented above. The customer-supplier process category supports transition of the software to the customer and its correct operation and

use. Together with various maintenance activities, these processes are not covered well by the product line approaches.

Generic Product Line Process Framework

The comparison of software product family process frameworks is based on a Generic Product Line Process Framework that is described in this section. The generic framework consists of process categories for product line management, domain engineering, application engineering, and third party product acquisition.

Corresponding to the traditional product process framework of Figure 1, the Generic Product Line Process Framework reflects creation of systems that are composed of four layers: system, product, platform, and component.

Components and 3rd party products are parts of a whole. They may be used as building blocks of any of the upper layers. Platforms have a double role: from a product viewpoint, they are components as they are integrated with some application functionality to build products. From the component viewpoint, platforms are similar to products as they typically consist of several components that have been integrated together. Finally, systems are solutions that consist of several products.

The process categories and their relations to each other and to created work products are illustrated in Figure 2. Compared to the previous model of Figure 1, this model replaces the Component/Platform Engineering process category with domain engineering, which may produce reusable assets for all levels of the layered systems. As domain engineering builds competence on the application area, domain engineering can give input to the portfolio management and management of 3rd party products. Domain engineering also interacts directly with the application engineering process groups.

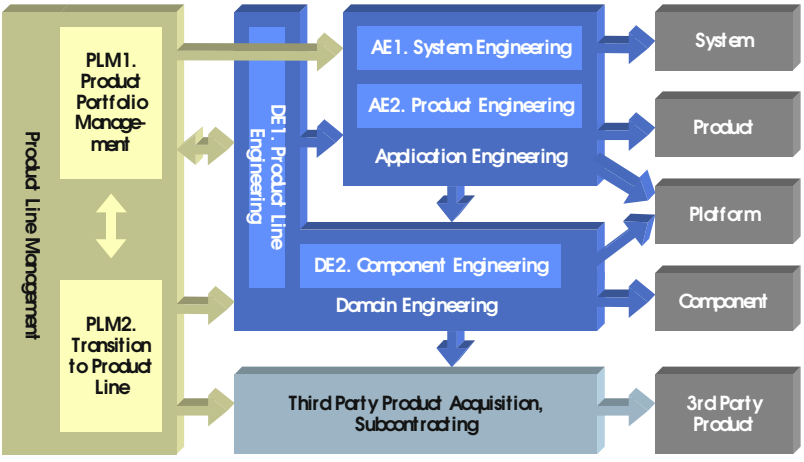


Fig. 2. Process categories of the proposed Generic Product Process Framework. The central part of the figure represents the actual Software Product Line Process.

Activity Groups

Product Line Management (PLM)	The product line management process category contains activities related to establishing and managing the product line.
PLM1. Product Portfolio Management	Creates visions and requirements for new products. This includes gathering requirements from the customers, market research, and technology research..
PLM2. Transition to Product Line	The transition process is temporary and not necessary after the product line infrastructure has been established. The transition process includes organizational planning and planning for competence creation. Reuse maturity assessment may be used to determine the organization's current reuse capability [Adoption93].
Domain Engineering (DE)	Domain Engineering is the activity to produce reusable assets. Domain Engineering is essentially orthogonal to the layered system-product-platform architecture and supports producing reusable components for all of the layers. Domain Engineering can play two roles: Product Line Engineering and Component Engineering. In the comparison, however, Domain Engineering is treated as a single activity group.
DE1. Product Line Engineering	Product Line Engineering is the variation of Domain Engineering for the entire product family. Product Line engineering concentrates on analysis of concepts common to all applications and design of common architecture for the complete product line.
DE2. Component Engineering	Component Engineering is the variation of domain engineering for a specific area of functionality or knowledge. Typically these areas represent the organization's technical core competencies. The resulting assets may be reused in all levels of the system-product-platform hierarchy.
Application Engineering (AE)	The term Application Engineering refers to the activities that generate new applications utilizing the assets created by Domain Engineering. In our terminology, there are two types of applications that have different creation processes: products and systems.
AE1. System Engineering	System Engineering is the activity to create systems utilizing reusable assets. Systems are solutions that consist of several products. Based on system requirements, System Engineering develops systems by integrating products.
AE2. Product Engineering	Product Engineering is the activity to create products utilizing platforms, components, and other reusable assets. Products may be supplied directly to the end-

customers or integrated to compose systems.

Third Party Product Acquisition and Subcontracting (TPS)

This activity group creates 3rd party product by acquisition of COTS components or through subcontracting. As the components produced by the Component Engineering activity, 3rd party products may be used in all levels of the layered systems.

Comparison

This comparison illustrates the coverage of the source frameworks compared to the Generic Product Line Product Process Framework. The comparison also maps the activities of the compared frameworks to the common terminology defined by the generic framework. The activity groups listed above are refined to consist of individual activities that make the rows of the comparison matrix. The columns represent different product line process frameworks. Their individual activities are distributed within the column to match the activities of the generic framework on the right column.

Table 3 shows an overview of the mapping without the names of the individual activities from the compared frameworks. The purpose of this overview is to illustrate which activities of the generic framework have been addressed by each of the compared frameworks.

Table 4 is an extract of the complete mapping to further illustrate details related to the Domain Engineering activity group. Note that RSEB defines two variations of domain engineering: Application Family Engineering and Component System Engineering. This separation corresponds to Product Line Engineering and Component Engineering activities of Figure 2.

For further details on domain analysis techniques, comparisons of plain domain analysis techniques have been published by Arango [Arango93] and by DeBaud and Schmid [DeBaud98].

Table 3. Coverage of compared SW product line process frameworks. One asterisk indicates some correspondence and two asterisks indicate good match with the activity named in the left column.

	SEI FSPLP	Synthesis, DsE	RSEB	SPICE, NRC SPF
PLM1. Product Portfolio Management				
Product Line Scoping	**	**	**	
Domain Management	**	**	**	
PLM2. Transition to Product Line				
Develop Organizational Strategy	**		**	*
Model Current Process	*	*	**	
Develop Product Line Process	*	**	**	*
Implement Product Line Process	**		**	*

Develop Metrics			**	
DE. Domain Engineering				
Domain Scoping	**	**		
Domain Analysis	**	**	**	*
Domain Verification		**		
Mine Assets	**			
Domain Design	**	**	**	*
Architecture Evaluation	**			
Domain Implementation		**	**	**
Integration and Testing	*	**	**	**
Domain Support	**	**	*	*
AE1. System Engineering				
Analyze Requirements				*
Design				*
Implement				*
Integrate and Test				*
Package				*
Supply				**
Support	*			**
AE2. Product Engineering				
Analyze Requirements	*	*	**	**
Design	*	*	**	**
Implement		*	**	**
Integrate and Test	**	**	**	**
Package			*	*
Maintain				**
TPS. Third Party Product Acquisition, Subcontracting				
COTS Utilization	**			*
Develop and Implement Acquisition Strategy	**			
Subcontractor Management				**

Table 4. Detailed mapping of activities within Domain Engineering activity group.

	SEI FSPLP	**Synthesis, DsE**	**RSEB: Application Family Engineering**	**RSEB: Component System Engineering**	**SPICE, NRC SPF**
Domain Scoping	TMP2. Product Line Scoping	DE.1. Domain Management			
		DE.2.1. Domain Definition			

Domain Analysis	SEP1. Domain Analysis	DE.2.2. Domain Specification	AFE1: Analyzing requirements that have an impact on the architecture	CSE1: Capturing requirements focusing on variability	ENG.1 Develop product requirements and design
			AFE2: Performing robustness analysis	CSE2: Performing robustness analysis to maximize flexibility	ENG.2 Develop SW requirements
Domain Verification		DE.2.3 Domain Verification			
Mine Assets	SEP2. Mining Existing Assets				
Domain Design	SEP3. Architecture Exploration and Definition	DE.2.2.4 Product (Family) Design	AFE3: Designing the layered system coordination	CSE3: Designing the component system	ENG.3 Develop SW design
Architecture Evaluation	SEP4. Architecture Evaluation				
Domain Implementation		DE.3.1. Product (Family) Implementation	AFE4: Implementing the architecture as a layered system	CSE4: Implementing the component system	ENG.4 Implement SW design
	SEP5. COTS Utilization				
Integration and Testing	SEP6. Software System Integration	DE.2.3 Domain Verification	AFE5: Testing the layered system coordination	CSE5: Testing the component system	ENG.5 Integrate and test SW
		DE.4.1 Domain Validation		CSE6: Final packaging of the component system for reuse	ENG.6 Integrate and test product
Domain Support	OMP3. Training	DE.4.2 Domain Delivery			
	OMP5. Launching and Institutionalizing a Product Line				

	TMP1. Data Collection, Metrics and Tracking	DE.3.2. Process Support Development	TRA6: Continuous process improvement		
	TMP3. Configuration Management				ENG.7 Maintain product and SW

Summary and Outlook

We have presented a Generic Product Line Process Framework and compared four publicly available product process approaches with the help of this generic model. The developed framework reflects the product structure of our industry and the compared product family process frameworks represent viewpoints that we consider important.

The comparison shows that the coverage of actual software engineering activities is rather complete by all of the compared frameworks. Deficiencies exist in management and other supporting categories and in the acquisition of 3rd party products. The system engineering field is only covered by NRC SPF. The SEI FSPLP covers all the other categories well. The weaknesses of Synthesis are the transition process and the 3rd party product acquisition process but it has the best coverage of domain engineering activities. RSEB does not cover 3rd party product acquisition. NRC SPF has the best coverage of customer support and maintenance activities but lacks several of reuse-oriented activities.

The first public version of the comparison is based on the work at Nokia Research Center. Further developmend of the model is to continue in an European ESAPS (Engineering Software Architectures, Processes and Platforms for System-Families) project during 2000-2001 [ESAPS].

Acknowledgements

The authors would like to thank prof. Jukka Paakki for his comments on the manuscript.

References

[Adoption93] Reuse Adoption Guidebook, SPC-93051-CMC, Software Productivity Consortium, Herndon, VA, 1993.

[Arango94] G. Arango, Domain Analysis Methods, in Software Reusability (W. Shaefer, R. Prieto-Diaz, and M. Matsumoto, eds.), Ellis Horwood, 1994.

[Campbell99] Grady H. Campbell, Jr., Reuse-driven Process Improvement. The first European Annual Conference on Reuse, London, April 1999.

[Clements99] Clements P, Northrop L, et.al., A Framework for Software Product Line Practice – Version 2.0, SEI, July 1999, (http://www.sei.cmu.edu/plp/framework.html).

[DeBaud98] Jean-Marc DeBaud and Klaus Schmid, A Practical Comparison of Major Domain Analysis Approaches - Towards a Customizable Domain Analysis Framework, In Proceedings of the Tenth Conference on Software Engineering and Knowledge Engineering, 1998.

[ESAPS] Engineering Software Architectures, Processes and Platforms for System-Families, European EUREKA/ITEA project, (http://www.esi.es/esaps/).

[Jacobson97] Jacobson I, Griss M, and Jonsson P, Software Reuse. Architecture, Process and Organization for Business Success, ACM Press, New York, June 1997.

[Känsälä99] Känsälä, Kari . Practices for Managing a Corporate-wide SPI Programme, European SEPG Conference, Amsterdam, The Netherlands, 7-10 June 1999.

[Nyström97] Nyström T, Comparison of Software Reference Processes Definitions, Master's thesis, HUT, 1997. 66 p.

[RSP93] Reuse-driven Software Processes Guidebook, SPC-92019-CMC, Software Productivity Consortium, Herndon, VA, November 1993.

[SPICE96] SPICE, Software Process Assessment Part 5: An assessment model and indicator guidance ISO/IEC/JTC1/SC7/WG10/N111, V2.0, October 1996, (http://www.sqi.gu.edu.au/spice/).

Issues Concerning Variability in Software Product Lines

Mikael Svahnberg[1] and Jan Bosch[2]

[1] University of Karlskrona/Ronneby
Department of Software Engineering and Computer Science,
S-372 25 Ronneby, Sweden, Mikael.Svahnberg@ipd.hk-r.se
URL: http://www.ipd.hk-r.se/msv/
[2] University of Karlskrona/Ronneby
Department of Software Engineering and Computer Science,
S-372 25 Ronneby, Sweden, Jan.Bosch@ipd.hk-r.se
URL: http://www.ipd.hk-r.se/jbo/

Abstract. Product-line architectures, i.e. a software architecture and component set shared by a family of products, represents a promising approach to achieving reuse of software. Several companies are initiating or have recently adopted a product-line architecture. However, little experience is available with respect to the evolution of the products, the software components and the software architecture. Due to the higher level of interdependency between the various software assets, software evolution is a more complex process. In this paper we discuss issues regarding variability that may help or cause problems when designing solutions for managing variability.

1 Introduction

In Sweden today, many companies already employ object-oriented techniques, such as design patterns and object oriented frameworks, and many are prepared to take the next step towards wide-scale reuse of software. A logical next step is software product lines, in which components and architecture can be reused over a number of applications. The software product-line defines a software architecture shared by the products and a set of reusable components that, combined, make up a considerable part of the functionality of the products. Much of the research efforts today regarding product line architectures is directed towards the initiation of a product line, and as a consequence, its evolution is not as well studied. One of the major issues regarding evolution is variability, i.e. how the product line allows for and facilitates the differences between the products in the product line. This paper presents and discusses some of the problems and issues that are relevant for any scheme that proposes to handle variability in software product lines. Moreover, we discuss the techniques available for introducing variability into the software product line.

The contribution of this paper is, we believe, that it presents the forces that may help or complicate the matter when selecting a technique to implement

F. van der Linden (Ed.): IW-SAPF-3, LNCS 1951, pp. 146–157, 2000.

variability. By increasing our understanding of software product line evolution as a result of incorporating new products and evolving the existing products, more efficient and effective support for variability can be provided.

The remainder of this paper is organized as follows. In the next section, we present our terminology, in order to help the understanding of the rest of the paper. In section 3, we present some observations made during a number of case studies regarding product line development. In section 4, we present techniques available for implementing variability, and discuss these with respect to where they are applicable. Related work is presented in section 5, and the paper is concluded in section 6.

2 Our View of Products, Architecture, and Software Product Lines

We define a software product line as consisting of a software product line architecture, a set of reusable components and a number of software products. The products can be organized in many ways, and the architectural variations are organized accordingly. A software product line architecture is a standard architecture, consisting of components, connectors, and additional constraints, in conformance to the definition given by Bass et al. [Bass et al., 1998]:

> 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. [Bass et al., 1998]

The role of the software product line architecture is to describe the commonalities and variabilities of the products contained in the software product line and, as such, to provide a common overall structure.

A component in the architecture implements a particular domain of functionality, for example, the file system domain, or the network communication domain. Components in software product lines, in our experience, are often implemented as object-oriented frameworks. A product is, thus, constructed by composing the frameworks that represent the component in the architecture.

We define a framework to consist of a framework architecture, and one or more concrete framework implementations. This interpretation of a framework holds well in a comparison to Roberts' and Johnson's [Roberts and Johnson, 1996] view, in which a white box framework can be mapped to our framework architecture, and a black box framework consists of several of our framework implementations in addition to the framework architecture.

The products in the software product line are instantiations of the product line architecture and of the components in the architecture. There may also be product-specific component extensions and, generally, product-specific code is present to capture the product-specific requirements.

3 Characteristics of Evolution

In previous studies [Svahnberg and Bosch, 1999a,Svahnberg and Bosch, 1999b] we have found several development characteristics that influence how variability is handled in software product lines. Below, we present and discuss these traits further.

The structure is static. This may seem to have very little to do with variability, but the fact that components stay in a particular place in a product line architecture makes it possible to rule out some types of variability, i.e. where the components must cope with completely new interfaces. The reason for why a software product line keeps a more or less static architecture is, we believe, that there are so many products that depend on a particular architecture - indeed, the entire development organization may depend on a particular architecture - that it is very costly to modify the overall structure of all systems. This means that not only do most products share the same overall architecture, this architecture remains the same as the products evolve as well. The implications this has on variability is that components do not necessarily have to be able to adapt to new situations, i.e. to be placed in new environments with new, unknown peers to operate with, and new connections to previously unknown components. It is sufficient that they can manage interaction with the set of components that they were originally designed to work with. Instead, effort can be put into ensuring that the components can manage variants of other components, i.e. that the components they interact with can be instantiated differently, as discussed below.

Variability on component-level is handled by selecting component implementations. There exists a number of supporting techniques to handle variability on the component-level. For example, the configuration management normally works best with full components, or rather implementations of component interfaces. This means that different products can use a simple configuration file to get the desired functionality by checking out entire subsystems, i.e. component implementations, from the source code repository. This type of variability is thus handled well with today s techniques.

Component interfaces evolve. If we assume the model where a component consists of an abstract interface and one or more concrete implementations of the interface, this implies that products are extended by adding new component implementations. Such new component implementations may be implementations of for example new standards, but may also include completely new functionality. Each new implementation in general implies changes to the abstract interface. A reason for this is that it is practically not possible to cover all future aspects of an interface from the beginning. To avoid this problem would require that all future implementations are analyzed and in effect designed before the product line is first implemented. This is of course not possible. Instead, the interfaces are adjusted as the evolution follows its natural course.

Previously, we stated that components can be designed with the assumption that the connections to other components will be fairly static. However, the evolution of the interfaces speaks against this advice. Changes in the interface can concern many levels, of which the most obvious are:

- Syntactic change, a method or class changes name, causing a simple search-and-replace.
- Semantic change, a method or class changes behavior. This may have considerable ripple-effects, and it may not be enough to examine only the place where the method/class is used. Since the impact of the change needs to be understood in order to find all places where other components may be depending on a certain behaviour, this type of changes cannot be done automatically.
- Growth of the interface. The interface can be extended with new methods or classes. In general, this does not have to imply anything on the older implementations, but if the growth results in a framework overlap [Mattsson and Bosch, 1999], measures must be taken against this.
- Reduction of the interface. Likewise, the interface may shrink, normally because functionality is broken out into a separate component. As with semantic changes, there may be ripple-effects of this.

Component implementations depend on parts of other components. Normally, the components in a product line architecture, or indeed any architecture, are conceptual entities that to the designers seems to be a coherent set of functionality. So is, for example, a component TCP/IP a natural component for a network-enabled product. As long as storage and execution space is ample this is not a problem, but as soon as one of these resources are in a short supply, you will want to scale off anything that is not needed in a particular product. It is then the other components and their implementations that decide on what is needed of a particular component. What is needs is thus to be able to, per product, configure a component so that only the desired functionality is included. This may involve restructuring the component architecture, which may not be trivial. This problem is also discussed by [Österbye, 1999].

On one hand, one can say that this is the very essence of variability, and on the other hand, one would want to reduce the problem to one that can more easily be solved. The latter can be achieved if the component can be split into several smaller components. This solution also suggests that the most logical components are not always the correct, and that perhaps components are not the lowest granularity desired. Rather, what is desired is feature-sets, that together comprise a logical component.

Evolution follows paths that are not easy to predict. The general guideline is usually that a list of predicted features spanning at least five years should be made when designing a software product line [Macala et al., 1996]. In many cases, the market moves to fast for this to work; products that are state-of-the-art today are not very exciting five years in the future. What this implies is that

the functional evolution can not be assumed to follow any designed or planned paths. On the technical side, one can assume that new products will have more or less the same architecture, and one can assume that the majority of changes are in the order of creating new component implementations, but modifications inside the components are simply impossible to predict. This implies that components can be designed (a) to incorporate any type of change, something which many agree is not desirable (e.g. [Jacobson et al., 1997]), or (b) to easily incorporate the planned changes without closing the door for other, unplanned, modifications.

4 Supporting Techniques

4.1 Levels of Variability

Our experience is that variability occurs at different levels in the design. Specifically, variability occurs on the product-line level, the architecture level, the component level, the sub-component level, and on the code level.

- *Product Line level.* This level is concerned with how different products in the product line varies.
- *Product Level.* At the product level, the variability is concerned with the architecture and choice of components for a particular product.
- *Component level.* On this level, the variability consists of how to add new implementations of the component interface, and also how these evolve over time.
- *Sub-component level.* As stated earlier, a component consists of a number of feature sets. On the sub-component level these feature sets are selected to create the component for a particular product.
- *Code level.* The code-level is where evolution but also most variability between products actually take place.

4.2 Available Techniques

To implement variability into the product line and the components, we have the means suggested by [Jacobson et al., 1997], namely:

- *Inheritance*, is used when the variation point is a method that needs to be implemented for every application, or when an application needs to extend a type with additional functionality.
- *Extensions* and extension points, is used when parts of a component can be extended with additional behaviour, selected from a set of variations for a particular variation point.
- *Parameterization*, templates and macros, are used when unbound parameters or macro expressions can be inserted in the code and later instantiated with the actual parameter or by expanding the macro.

- *Configuration* and Module Interconnection Languages, are used to select appropriate files and fill in some of the unbound parameters to connect modules and components to each other.
- *Generation* of derived components, is used when there is a higher level language that can be used for a particular task, which is then used to create the actual component.

In addition to these techniques, we would like to add *"ifdefs"*, which are used to at compile-time select between different implementations in the code.

By *parameterization* is meant that a component or a class is given some initial values that regulates how it is to work. This can be things like a base value, or even a parameter type. A related concept is code generation. In parameterization the source code is written in traditional ways, and finalized using some parameters, either at compile-time or at start-up time. In code generation, the source code is created as a consequence of a number of choices of the software engineers, and is thus hard-coded to a particular set of parameters.

Two of the techniques related to parameterization are *templates* and *ifdefs*, both language constructs in C++. Templates is a mechanism by which the choice of types to operate on is delayed until a class is used, rather than when it is created. For instance, we can have a class LinkedList , that is written so that it uses a template. When the linked list is later used in a program, the template class type is replaced with a particular type, for instance Word , which thus creates a linked list of words. With the exception of some performance benefits, the same functionality can be achieved using inheritance and abstract base classes. Preprocessor directives is another feature of C++, which enables a more fine-grained configuration management. Parts of the source code can be surrounded by so called ifdef statements, which means that they can at compile-time be included or excluded from the compiled code.

By *configuration* is meant the process in which source code is selected from a code repository and put together to form a particular product. Module Interconnection Languages is one way of describing configurations. Once the correct files are selected, some of the parameterization can also be performed by more advanced configuration management tools. The final configuration is performed by the compile utility, using for instance make files.

Inheritance is the standard object oriented way of extending a class with more behavior by inheriting from it and adding the extra behavior. This allows for variability by reusing everything that is common to the new application and only replace or extend with those things that differ. Extensions and extension points is a more planned way to use inheritance, where several implementations (i.e. classes that are inherited and implemented from an abstract base class) can coexist in one application.

Many of the techniques introduced by [Jacobson et al., 1997] can also be found in *Design Patterns* [Gamma et al., 1995]. A design pattern is a proven design solution for a particular problem that has been used in many applications. It has long been recognized that using design patterns improve the structure of software, and thus also improves on maintainability and reusability. There

exists design patterns to provide Extensions and extension points, but also for parameterization. Design patterns rely heavily on the techniques of inheritance and extensions, but also, to some extend, on parameterization.

4.3 Applicability of Techniques

The levels of variability provides, in a way, a development method, starting from the large picture and moving down to the source code level. Below, we discuss the levels of variability from the perspective of creating a product, and what the expected variability is on each level. We also discuss how well the available techniques can solve the expected variability issues on each level.

Product Line Level. On the product line level, what is done is to select a set of components for a product from a component repository. Product specific code is also either generated or selected from a similar repository. Variability on this level is concerned with how products differ, i.e. what components different products use and what product specific code (PSC for short) that is used.

Product Level. On the product level, the components are fitted together to form a product architecture, and the PSC is customized for the particular product variation. Variability issues on this level are a) how to fit components together, b) how to cope with evolving interfaces, and c) how to extract and/or replace parts of the PSC.

Component Level. As we now have selected the components and connected them, we are on this level concerned with selecting what particular component implementations to include into the product. As stated earlier, we view a component as an abstract object oriented framework with a number of framework implementations. This level is where the set of framework implementations are selected. These framework implementations are also connected to the abstract framework, and lastly, the PSC is bound into not only the abstract framework, but also into the concrete implementations. Variability issues here are how to enable addition and usage of several component implementations, and how to design the component interface in such a way that it survives the addition of more concrete implementations. This is slightly different than the evolving interface issue on the product level, since there the concern was how to cope with the interfaces from the outside of the component. On this level, the concern is how to cope with the evolving interfaces from the perspective of the various component implementations.

Sub-component Level. We previously observed that functionality spans a number of components, and depending on the configuration of one component, other components are affected as well. To avoid dead code, the parts of components that are not used should be removed. Note that removing a particular feature causes modifications to all of the component implementations, since

they all implement the feature. The variability issue on this level is thus how to remove or add parts of a component where each part spans all component implementations.

Code Level. On the code level, everything stated above must be put into place. If the previous steps have been followed, all that remains to do is to make sure that the provided class interfaces match the method calls performed, i.e. the required interface. This is probably the hardest variability challenge of all. As components and classes evolve, so do their interfaces. These interfaces exist in more than one component implementation, and are used by more than one component. Moreover, each product may use a separate version of the component implementation, and thus a separate version of the interface. All of this must be put together on this level.

The Variability Mechanisms. Having defined the steps by which a product is instantiated from a software repository, this section discuss how the variability mechanisms presented in section 4.2 can be applied to the different steps.

Configuration. In the early stages, configuration plays a significant role. On the *product line level*, it is used to select the components and the PSC from a code repository, albeit this requires that there is a clear separation between generic and product specific code.

On the *product level*, the selected components are connected together, for instance by using tools such as module interconnection languages.

Configuration on the *component level* is concerned with selecting the actial concrete implementations to include into the product. Logically, these are usually seen as part of a single component, but from the view of configuration management, they are often seen as subsystems.

If the components have been designed as a collection of disjoint sub-components, configuration management can be used to select the specific parts of the components to include on the *sub-component level*, but otherwise configuration management plays, at this stage, a less significant role. Configuration management is also practically useless on the *code level*, since it is to coarse-grained.

Ifdefs and Parameterization. Almost hand in hand with configuration management goes ifdefs and parameterization. In a way, ifdefs can be seen as a more fine-grained configuration management tool, and parameterization is only a special case of this, were no code is excluded from the compiled binary, as is the case with ifdefs. A general disadvantage of using ifdefs and parameterization is that as the number of products increase, this soon becomes unmanageable. Consider, for example, if the code base is built up using ifdefs to distinguish between products and a new product is introduced. This means having to find and modify all the ifdef statements throughout the entire code base. Parameterization is used in similar ways, with the exception that the unwanted source code remains in the compiled binary, and is simply not executed.

On the *product line level*, ifdefs is the way to remove unwanted PSC, if it is not cleanly enough separated to use configuration management tools instead. On the *product level*, these two techniques can be used to connect components to each other, but this allows for only a very static way of connecting components.

On the *component level*, parameterization can be used to select the component implementations to use, but this requires that the abstract component is aware what particular implementations that exist. Moreover, the compiled binary will contain dead code , i.e. code that is never executed, depending on what configuration of component implementations that is selected. Ifdefs have a number of usages on this level; they can be used to a) insert the PSC into the component implementations, b) change the component interface depending on the configuration of the component, and c) to select the set of component implementations to include into the product. The disadvantage of all of these usages is that they are not scalable. Case (a) and (c) increase in complexity as the number of products and the number of component implementations, respectively. Case (b) is also depending on the number of component implementations, but with the additional disadvantage that every component implementation and all other components that use these component implementations needs to be drastically modified for every interface change. However, they do need to be modified anyway, and using ifdefs can ensure that older products can still be generated from the same code base. Parameterization and ifdefs are used in a similar fashion on the *sub-component level*, but the disadvantage is that parameterization results in dead code, and the complexity of the code increases, as stated above. For the same reasons, we discourage usage of these two techniques for both the sub-component level and the *code level*.

Inheritance and Extensions. These two techniques plays a substantial part on all the levels of variability, mainly because they can be used together with both configuration management, parameterization, and more run-time oriented variation mechanisms. Inheritance and extensions provide a way to divide the source code into several files, which makes it possible for the course-grained file-based configuration management tools to select and deselect individual extensions. As the importance of configuration management decreases in the later levels of variability, the more fine-grained technique of parameterization increases in importance, and inheritance then supports variation on the class level. When parameterization decreases in usefulness, inheritance and extensions step up as a major technique in their own right, in the form of Design Patterns.

Templates. Using templates instead of inheritance yields the same benefits, plus a slight performance increase. The drawbacks of using templates is that the generic code needs to be parameterized with the class names of the PSC, and the interface becomes implicit. However, this is sometimes an advantage, since it makes it possible to hide interface changes. Another limitation becomes evident on the sub-component level, because more than one extension is allowed to be present in a system at one given time, and this is not technically possible when using templates. With templates is that you get a one-to-one mapping between

a using class, e.g. a linked list, and the class used to instantiate the template, e.g. a word-class.

Generation. Code generation is a technique that, to our knowledge, is not used very often in industry. Its main usage would be on the product level, where the code from the software repositories needs to be instrumented with product specific code, and in particular glue code to connect the components to each other. On the other levels, it becomes a more philosophical question: should you create a tool that generates source code, or should you write the source code directly?

4.4 Analysis of Techniques

By examining the usage of the techniques further, we see that configuration management plays a substantial part in the higher levels, i.e. the product line, product, and component level, after which the usage dwindles to practically nothing. A similar curve is generated by ifdefs and parameterization, albeit slightly less usable in the higher levels, and useful slightly longer. As the usage of configuration management and parameterization decreases, the usage of inheritance and extensions increase, and in parallel with this, so does the use of templates.

Unfortunately, this is often not the case in industry. Instead, variability is in many cases implemented using parameterization (i.e. templates and ifdefs) for all levels of variability. Tools designed for code level variability are thus used on all the levels, which causes many troubles with respect to understanding what is happening, and how to maintain the product line.

5 Related Work

[Jacobson et al., 1997] is probably the main reference regarding variability today. The book discuss all the topics connected to software reuse, of which variability is a major issue. The book focus mostly on how to implement variability into a newly created product or product line, and does not cover evolutionary aspects as extensively. Our work is based on the techniques identified to achieve variability, and presents an overview of how these can be used in an evolving product line. Object oriented frameworks has been a recognized way of achieving software reuse for quite some time now, and naturally the discussions also concern variability. For instance [Roberts and Johnson, 1996] present what they call hot spots , i.e. places where the framework is likely to change for every new release and usage.

The two major techniques for variability as identified are configuration management and design patterns. Configuration management is dealt with extensively in [Conradi and Westfechtel, 1998], presenting the common configuration management tools of today, with their benefits and drawbacks. Design patterns are discussed in detail in [Gamma et al., 1995], where many of the most used design patterns are also presented.

Academia has come up with a number of techniques that are not used very much in industry. Some of the more interesting are Aspect-, Feature-, and Subject-oriented programming. In Aspect-oriented programming, features weaved into the product code [Kiczalez et al., 1997]. These features are in the magnitude of a few lines of source code. Feature-oriented programming extends on this concept by weaving together entire classes of additional functionality [Prehofer, 1997]. Subject-oriented programming [Kaplan et al., 1996] is concerned with merging classes developed in parallel to achieve the combined functionality of both. Although interesting from a technical perspective, neither of them are used in industry, and they all require more discipline from the programmers. They all claim to improve readability of the source code by extracting sets of behaviour from the product code, but we hold doubt whether this technique helps understandability of the code actually executed.

6 Conclusions

A cost-effective management of variability is one of the key issues for a successful product line. In contrast to what is usually said (e.g. [Macala et al. 97]), it is not practical to map out all the planned products that is to be fit into the product line with a five year span. Although predicting technology changes and other developments is important, a substantial part of the new requirements on the software product line cannot be predicted. The consequence of this is that the product line is not designed for all the future products, but rather for the products that exist today. To handle variability well is then to handle the variability required of the software product line today. There are a number of characteristics that distinguish product line development from traditional software development. This paper presents a number of these characteristics, based on previous industry case studies. These characteristics are useful when selecting between different techniques to introduce variability into the software product line. Moreover, we have noticed that variability is introduced on different levels of the product line, i.e. product line, product, component, sub-component, and code level. The challenges on these levels differ, and hence so does the most suitable solution. We discuss the commonly used solutions for each level of variability, presenting the benefits and drawbacks of each solution on each level.

As part of future work, we intend to continue to study variability management in industrial contexts. based on these results, we intend to develop guidelines and techniques to improve on the problems discussed in this paper.

References

[Bass et al., 1998] Bass, L., Clements, P., and Kazman, R. (1998). *Software Architecture in Practice*. Addison-Wesley, New York, NY.

[Conradi and Westfechtel, 1998] Conradi, R. and Westfechtel, B. (1998). Version models for software configuration management. *ACM Computing Survey*, 30(2):232 – 282.

[Gamma et al., 1995] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley Longman, Reading, MA.

[Jacobson et al., 1997] Jacobson, I., Griss, M., and Jonsson, P. (1997). *Software Reuse: Architecture, Process and Organization for Business Success*. Addisson Wesley, New York, NY.

[Kaplan et al., 1996] Kaplan, M., Ossher, H., Harrisson, W., and Kruskal, V. (1996). Subkecy-oriented design and the watson subject compiler. Position paper for OOPSLA'96 Subjectivity Workshop.

[Kiczalez et al., 1997] Kiczalez, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. (1997). Aspect-oriented programming. In *Proceedings of 11th European Conference on Object-Oriented Programming*, pages 220–242, Berlin, Germany. Springer Verlag.

[Macala et al., 1996] Macala, R., Stuckey, L., and Gross, D. (1996). Managing domain-specific, product-line development. *IEEE Software*, 13(3):57–67.

[Mattsson and Bosch, 1999] Mattsson, M. and Bosch, J. (1999). Composition problems, causes, and solutions. In Fayad, M., Schmidt, D., and Johnson, R., editors, *Building Application Frameworks*, chapter 20, pages 467–486. John Wiley & Sons Ltd., New York, NY.

[Österbye, 1999] Österbye, K. (1999). Vertical objects in a horizontal architecture: Design issues in a component based architecture for doculive. In *Proceedings of the Second Nordic Workshop on Software Architecture (NOSA'99)*.

[Prehofer, 1997] Prehofer, C. (1997). Feature-oriented programming: A fresh look at objects. In *Proceedings of ECOOP'97*, number 1241 in Lecture Notes in Computer Science, Berlin, Germany. Springer Verlag.

[Roberts and Johnson, 1996] Roberts, D. and Johnson, R. (1996). Evolving frameworks: A pattern language for developing object-oriented frameworks. In *Proceedings of PLoP-3*.

[Svahnberg and Bosch, 1999a] Svahnberg, M. and Bosch, J. (1999a). Characterizing evolution in product line architectures. In Debnath, N. and Lee, R., editors, *Proceedings of the 3rd annual IASTED International Conference on Software Engineering and Applications 1999*, pages 92–97, Anaheim, CA. IASTED/Acta Press.

[Svahnberg and Bosch, 1999b] Svahnberg, M. and Bosch, J. (1999b). Evolution in software product lines: Two cases. *Journal of Software Maintenance: Research and Practice*, 11(6):391–422.

A First Assessment of Development Processes with Respect to Product Lines and Component Based Development

Rodrigo Cerón[1,2], Juan C. Dueñas[2], and Juan A. de la Puente[2]

Department of Engineering of Telematic Systems,
Universidad Politécnica de Madrid
ETSI Telecomunicación, Ciudad Universitaria, s/n, E-28040 Madrid
ceron@dit.upm.es, jcduenas@dit.upm.es, jpuente@dit.upm.es

Abstract: One important line of research in the application of product-line approaches to the industrial field is the creation and adaptation of already known development processes to hold the activities required to build families of products, based on the usage of components. There are different development models in the literature, but each of them focus on a specific technique or approach, so in order to create an effective development process suited for that purpose, several of them must be joint. This article reviews a well-known development process, and evaluates it in the light of product lines development and also the creation and usage of components. This is achieved by comparison with other four processes, specially tailored to those activities. This is the starting activity of a larger effort driven towards the definition of a general development process, able to apply the best practices known to the creation of product lines with components.

Introduction

The current situation, as regards the development of complex software-systems, is the usage of architecture-centric approaches, supported by the usage of either in-house modelling formalisms, domain specific formalisms, such as SDL, or general description languages, such as UML.

---

[1] Prof. Rodrigo Cerón is a visiting professor from Universidad del Cauca, Popayán, Colombia, granted by contract OJ087/1999.

[2] This work has been partially developed in the project "Engineering Software Architectures, Processes and Platforms for System-Families" (ESAPS) ITEA 99005/Eureka 2023, and has also been partially funded by Spanish CICYT under the project "Integrated development for distributed embedded systems".

F. van der Linden (Ed.): IW-SAPF-3, LNCS 1951, pp. 158-167, 2000.

Once the modelling language problem has been solved, the focus turns to the process: how to apply the formalisms adequately in order to produce product-lines, what is the extent of the models, when to produce them and when to finish polishing, etcetera. The development process must be defined taking into account not only technical requirements such as integration and tool support, but also management ones, such as flexibility, repeatability, economy and "time to market".

Some key issues that companies developing software-intensive systems are pursuing are the techniques around the production of families of applications, and the usage of components in order to reduce make the development faster.

Current situation in the industry as regards product lines leads to the production of different systems in the family following an incremental approach, where a new product reflects the reuse of a previous one, as well as improvements to its functionality or performance. This approach allows the quick production of new applications, but provides a low support degree for previous versions. Variants convert into versions.

With respect to the usage of components, it is important to note that this strategy has been traditionally regarded as an "implementation activity". Since there was little or no support for reuse at analysis or design phases, because there was no design entities for components, the community developing components and groups promoting reuse at early phases of development had no communication between them.

The ESAPS project (EUREKA 2023, ITEA 9905) is trying to joint the best of both worlds by using components at the architectural level, in order to support the effective development process. This article reviews a well documented development processes in the second section, and compares it to three development process specially suited for product lines in third (PRAISE) and fifth section (RSEB and FORM), a component-based engineering in the fourth section (SELECT). Our attempt in the ESAPS project will be to populate this evaluation and to get a catalogue of activities that can be combined to get a general development process.

The Rational Unified Process Model

The Rational Unified process [2], [3] has been proposed by the Rational corporation, as a general model that holds "best practices" in the development of complex systems (not only software). The main forces behind its application are:

1. architecture-centric development: the efforts are guided and documented by the architectural models of the system, that act as a roadmap for all stakeholders. The architecture is used to understand the system, organise the development, foster reuse and evolve the system.
2. iteration: following the main stream in development processes, that regard the pure waterfall model as obsolete, the process is organises as a series of iterations holding nearly all kinds of activities (in more or less degree), that improve the system until the degree of coverage of requirements is adequate. Thus, a software product is developed in small and manageable steps.
3. use-case driven: the start of each iteration is the definition or focus on new use-cases (based on functional requirements) of the system. Gradually, the system gets more and more functions, added following these use-cases.

Following these main ideas for this kind of approach, the unified process repeats over a series of cycles, where each one concludes with a release of the product. And each cycle consists of four phases: inception, elaboration, construction and transition. Each phase terminates in a milestone and it is further subdivided into iterations (waterfall mini-projects).

The inception phase transform a good idea into a vision of the end product, the elaboration phase design the architecture, the construction phase build the product and the transition phase deliver the version of the product to their users. The product evolves through these phases until it is considered obsolete.

Workflow

The core workflows (Requirements, Analysis, Design, Implementation and Test) and its activities and related workers (roles) are shown in figure 1. Each phase is composed by the core workflows. The time advances from left to right and the arrows between the activities represent temporal relationships.

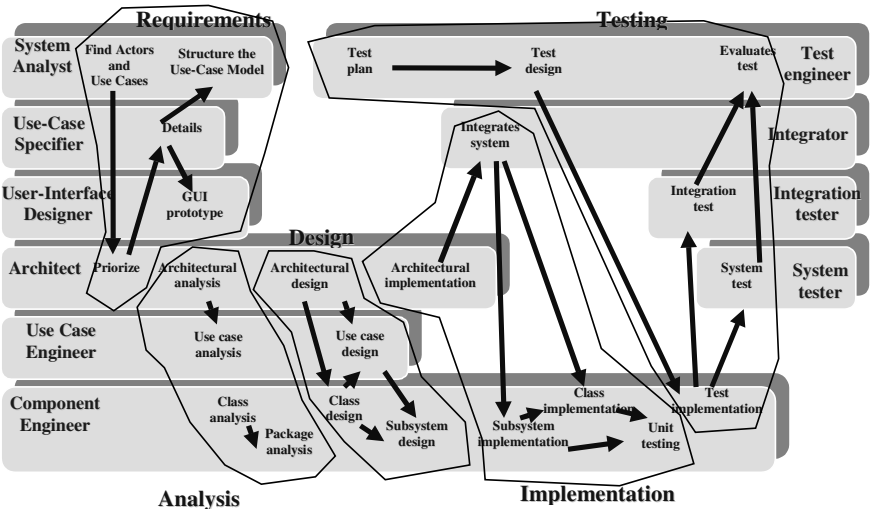


Fig. 1. Workflows in the Rational Unified process

Roles

The process proposes the usage of several main roles that can be specialised later. The basic roles are: System Analyst, Use-Case Specifier, User-Interface Designer, Architect, Use Case Engineer, Component Engineer, Test Engineer, System Integrator, Integration Tester and System Tester. One person can play one or various roles according to the workflow. The activities assigned for each role can be extracted from figure 2.

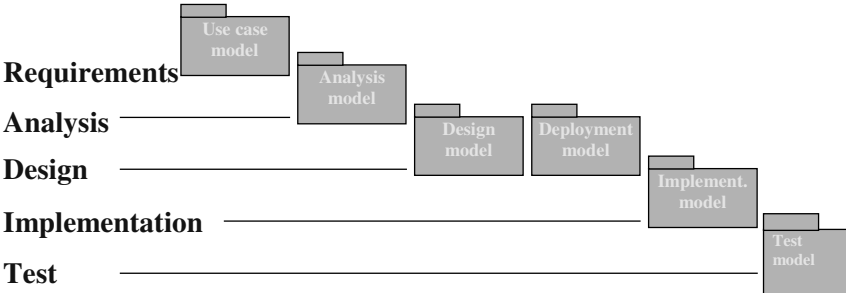


Fig. 2. workflows vs. models in Rational Unified process.

Products

In the Unified Process, we can distinguish six models (Use-Case Model, Analysis Model, Design Model, Deployment Model, Implementation Model and Test Model). All the models but the Test Model uses UML. The relation of core workflows and models is shown in figure 1.

The relationship between phases and deliverables is shown in table 1.

Table 1: workflows vs. deliverables in Rational Unified process.

Inception	Elaboration	Construction	Transition
• Feature list • First version of Business Model • First cut Models • First draft of candidates architectures description. • Possibly exploratory prototype • An initial risk list • The beginnings of a plan for the entire project • A first draft of the business case	• Preferably a complete business model • A new version of all models • An executable architectural baseline • An architectural description • Updated risk list • Project plan for the construction and transition phase • A preliminary user's manual • Completed business case	• Project plan for the transition phase • The executable software itself • All tangible pieces of information • An updated architecture description • Preliminary user's manual • Business case	• The executable software itself including installation software • Legal documents • Completed and corrected product release baseline (all models) • Completed and updated architecture description • Final manuals • Customer support references and web references

Being a general development process, that must be adapted to a specific company and application domain, no activities have been found in order to produce a product line. The usage of components is partially covered in the workflow by the activities performed by the "Architect" and " Component Engineer " role. We will discuss these two kinds of activities, comparing the process to other specific processes.

Product Lines Development Activities

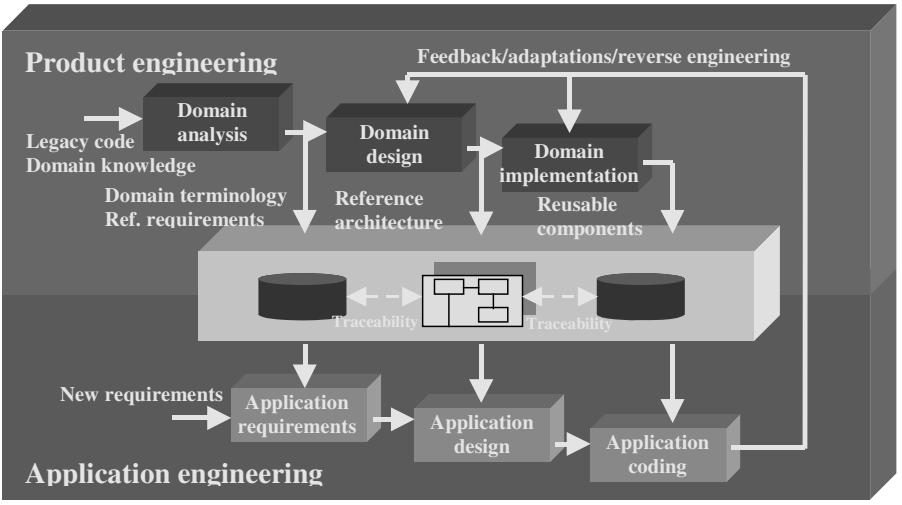


Fig. 3. the PRAISE development process model.

In order to evaluate the Unified Rational process with respect to product line activities, the PRAISE development process model is a help. This development process has been specifically created in order to define the activities and products that building product lines must include. This process model was created by the European project PRAISE, as an attempt to study the process of creation of product lines and derivation of single products in the family.

One way to assess the Unified Rational process is then to compare its workflow with PRAISE's, matching activities common to both of them. Unfortunately, the PRAISE model is still at the abstract level (it is a kind of meta-process), so there are no specific details about how to perform each of the activities. So, more than a comparison activity by activity, PRAISE can serve as a pattern in order to check product-lines development process conformance.

An important issue of PRAISE is that it has been defined following the waterfall schema (although some degree of iteration is supported by the arrow that includes feedback, adaptations, reverse engineering). For both the workflow and the tasks, see Figure 3.

The results of our comparison of the Unified Rational process with respect to the PRAISE process model are:

- the Unified Rational process does not support generic activities (product-lines) explicitly; there are no activities in order to derive products from reference architectures. In PRAISE, this is described with two main flows of work: the product engineering for families of products, and the application engineering for a single product.
- the Unified Rational process seems to be a "from-the-scratch" approach to the development of a single product; so no inputs appear, taking into account the feedback, legacy code, et cetera.

- the difference between single product and set of related products does not appear in the process description. Activities such as delivery of different products, definition of policy for replacements and some more should be added.
- in any case, the Unified Rational process seems to support better the application engineering flow, based on the usage of components (that could have been produced in the product engineering flow). The notion of component appears, and so the roles such as "architect" and "component engineer".

The iterative approach of the Unified Rational process, in any case, could hold the development of product lines, provided that the core functions or elements in the family can be packaged into a single product. Next iterations lead to the production of new applications. This approach also leads to the harvesting of the set of components that can be used in subsequent iterations (the "domain implementation" task in the PRAISE model). In any case, new roles must be added to the Unified Rational process, at least for domain analysis and domain design. An architect of applications may be do the domain analysis and domain design, for example.

Component-Based Development Activities

In order to assess the Unified Rational process, and following the same approach than with product lines, a specific development process based on components, has been chosen and studied. The SELECT development process [4] is now described. The process is iterative and product focused, and it is divided in two sub-projects: the "solution project" and the "component project". The first project aim to produce solutions by assembly of components and the second one aim to sow reusable services in the manner of component builders.

The high level workflow is shown in Figure 4 where BPM stands for Business Process Model. BPM can be considering an optional and highly recommended process, it is done prior to component and solution projects, and then the solution and component process are performed.

The basic solution team roles are Project Manager, Technical Co-ordinator, Executive Sponsor, Visionary, Developer/Senior Developer, Ambassador User, Adviser User, and Team Leader. Additional staff may perform the roles of Reuse Identifier, Testing Specialist, Human Factor Specialist, and Business Consultant. The basic component team roles are Reuse Manager, Reuse Librarian, Reuse, Assessor, Reuse Architect, and Component Developer. The component team can also make a call on solution and the additional team roles. The solution and component teams must be supported by Technical Infrastructure Team. The Technical Infrastructure Team roles are Technical Facilitator, Network Expert, Web Expert, and Capacity and Performance Manager. For each role, there is a general description in [4].

In the SELECT Process, we can distinguish eight models (Business Process Model, Use-Case Model, Class Model, Object Interaction Model, State Model, Component Model, Deployment Model, and Logical Data Model). All the models but the Business Process Model and Logical Data Model use UML. The Business Process Model use a notation adapted from CSC Catalyst. The Logical Data Model is optional and use a notation adapted and simplified from CCTA [5], this model is employed only in the case in which relational databases are used for data storage.

Compared to the Unified Rational process, there is a great difference in focus. In the Rational process, iteration applies to a central architecture, whereas in the SELECT approach, iteration aim to harvest the set of generic and reusable components.

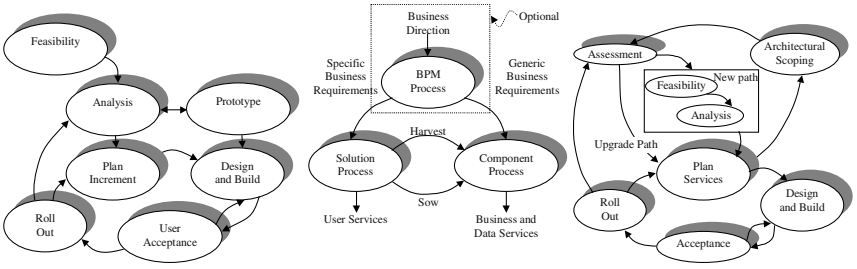


Fig. 4. the SELECT development processes.

In order to adapt the Unified Rational process more closely to the management of reusable components:

- the main improvement that SELECT can add to the Unified Rational process is the sowing process for the development of a component. When there are activities performed by the architect, an additional possibility appears: to browse and select previously developed components. In this way, the ideal case of development of a product just by "gluing" components is also supported; this is not clear with Unified Rational Process but it is possible to do.
- a new entry point should be added to the workflow, in order to implement the harvesting process, not driven by one use-case, but from generalisation from previous developed systems.
- explicit activities in the "architect" track must be added, specially for the identification and acquisition of components.
- activities in the "component engineer" track must be added, specially for the generation of reusable components.

The main strength with this approach is the concentration of the decision in two roles "architect" and "component engineer" but it may produce a non-repeatable process.

Reuse-Driven Software Engineering Business (RSEB) and Feature-Oriented Reuse Method with Domain-Specific Reference Architectures (FORM)

RSEB [7] and FORM [8] are suited for product lines because they define a systematic method that focuses on capturing commonalities and differences of applications in a domain and using analysis results to develop domain architectures and components.

In order to obtain a software engineering the RSEB process is divided into three categories: Component System Engineering, Application System Engineering, and Application Family Engineering. Application Family Engineering process determines how to decompose the overall set of applications into a suite of application systems

and supporting component system. Application System Engineering process selects, specializes, and assembles components from one or more component systems into complete application systems. Component System Engineering process designs, constructs, and packages components into component systems. It defines a series of roles (workers): Software Engineering Bussines Manager, Process Owner, Process Leader, Use Case Engineer (Superordinate Use Case Engineer), GUI Coordinator, Subsystem Engineer (Superodinate Subsystem Engineer), Use Case Designer (Superordinate Use Case Designer), Tester, Reuse Process Engineer, Reuse Support Enviroment Engineer, Facade Engineer, Architect (Lead Architect), Distribution Engineer, Component System Trainer, Component System Supporter, Component System Librarian, and Application Engineer [7].

The FORM method consists of two major engineering processes: domain engineering and application engineering (see Figure 5). The domain engineering process consists of activities for analyzing systems inside a domain and create reference architectures and reusable components based on the analysis results. The reference architectures and reusable components are expected to accommodate the differences as well as the commonalties of the systems in the domain. The application engineering process consists of activities for developing applications using the artifacts created in the domain engineering.

The results of our comparison of the Unified Rational process with respect to the RSEB and FORM process model are:

- both RSEB and FORM consider the development of a family of applications.
- the Unified Rational process does not take into account activities related with the study of commonalties of applications.
- the Unified Rational process considers the architecture as the central point in the development process but it seems to be derived for a single application because it does not consider activities such as domain analysis in FORM.
- the Unified Rational process can be adapted in order to support product lines adding roles (like the RSEB proposes) and activities in the core workflows (like the FORM proposes).

Conclusions and Further Work

The main conclusions that can be got about product line and component based development activities is that lacking of practical experience is the main problem to their widespread acceptance.

Some additional points where these approaches can improve are the provision of either conceptual and physical tools that support:

- the description of variant and common parts in the reference architecture, using UML. Some efforts have been reported [6], but we are still far from a clear design representation of common/variant parts, that does not interfere with the understanding of other design principles.
- the presence of components provided by different companies at the architectural level. Some attempts are the description of DCOM components, for example, by means of interfaces. However, a better working documentation, given as contracts of operations, should be provided.

- method guides to incorporate to the current in-house development processes, the harvesting activities in order to create components that the same company is able to reuse later. Especially, smooth paths for implementing product-line and component based activities within these processes.
- changes to the organisation, sometimes regarded as "business process reengineering" are still necessary to create a flexible organisation able to respond quickly to the market requirements.

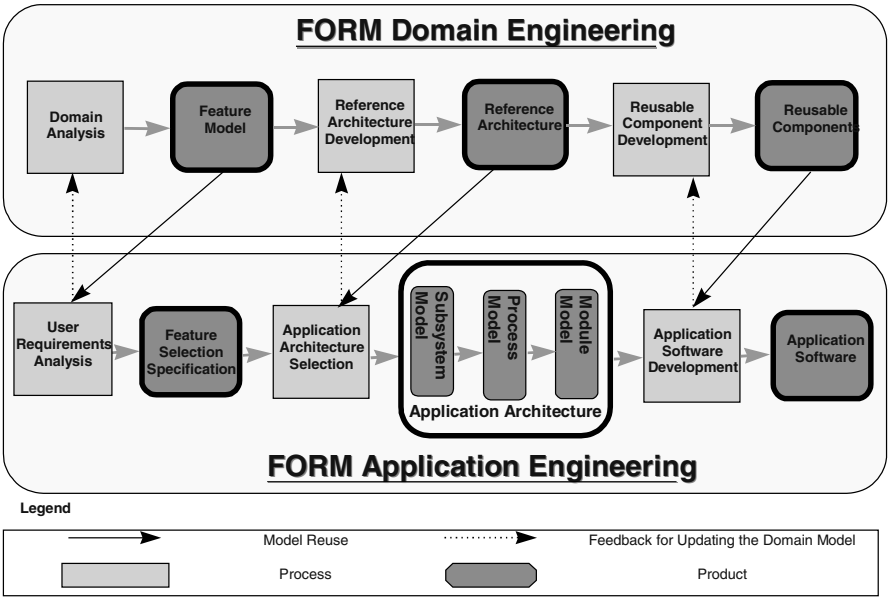


Fig. 5. FORM Engineering Processes.

Our future work in the project will be to produce the catalogue of common activities and the guides to adopt them in already running development processes, and the study of principles and guides to represent variability with UML, and to identify abstract components in a given architectural model.

Bibliography

[1] M. Shaw, D. Garlan, *Software Architecture. Perspectives on an Emerging Discipline,* Prentice Hall 1996.

[2] I. Jacobson, G. Booch, J. Rumbaugh, *The Unified Software Development Process,* Addison Wesley 1999.

[3] P. Kruchten, *A Rational Development Process*, Rational WWW site. http://www.rational.com/sitewide/support/whitepapers/dynamic.jtmpl?doc_key=334.

[4] P. Allen, S. Frost, *Component-Based Development for Enterprise Systems: Applying The SELECT perspective,* Cambridge University Press 1998.

[5] CCTA, *SSAMD Version 4 + reference Manual,* NCC Blackwell 1995.

[6] B. Keepence, M. Mannion. *Using Patterns to Model Variability in Product Families* , IEEE Software, 16/4, July 1999. IEEE Computer Society.

[7] I. Jacobson, M. Griss, P. Jonsson. *Software Reuse: Architecture, Process and Organization for Business Success*, Addison Wesley 1997.

[8] K. Kang, S. Kim, J. Lee, K. Kim, E. Shin, M. Huh. *FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures*, 1998

Evolution of Software Product Families

Jan Bosch[1] and Alexander Ran[2]

[1]Department of Software Engineering and Computer Science
University of Karlskrona/Ronneby
Sweden
[2]Nokia Research Center
5 Wayside rd., Burlington, Ma 01803
USA
Jan.Bosch@ipd.hk-r.se
alexander.ran@nokia.com

Abstract. Evolution in software product families is a difficult problem that is not well understood and studied insufficiently. In this article, we present a categorization of product family evolution, a discussion of the implications of architectural evolution and a summary of the discussion during the workshop.

Introduction

The session on evolution of architecture of software product families was one of the two special sessions of the workshop. The Workshop Program Committee introduced it into the program even though no papers were submitted to the workshop that would comfortably fit this topic. We all recognized the importance of change and evolution in design, development, and management of software product families, and we regarded IWSAPF-3 workshop as an opportunity to make progress in our joint understanding of the subject.

We got an assignment to prepare a position on the subject of evolution of software product families. We have presented our position as an introduction to a common discussion. In this paper we overview the main ideas of our introduction to the topic of evolution in software product families and mention some interesting points and ideas that were raised during the discussion.

Understanding Evolution and Change in Software Families

The products in a software product family are used by customers. The use of the software product generally leads to new requirements. These new requirements emerge reactively, because the users require additional features, or proactively, since the marketing responsible for the product identifies new features that allow for new uses of the product or that will result in increased market share. This process occurs in parallel for all products in a product family. New requirements do no just originate from the customers, but may also result from evolution of the technology used by the software product, e.g. hardware and third-party components and new standards.

F. van der Linden (Ed.): IW-SAPF-3, LNCS 1951, pp. 168-183, 2000.

Whenever new requirements emerge, the first decision that has to be taken is whether these requirements should be incorporated or that the software product should not develop into the niche indicated by the new requirements. Once it is decided that we should incorporate the requirements, the second decision that needs to be taken is whether the requirements have, or should have, effect on the product family as a whole, or that the effects can be restricted to the particular product at hand.

If the requirements are only relevant for the product at hand, incorporating the requirements is performed as part of the product-specific code and is similar to the ways of working in the traditional, one-system-at-a-time model. However, if we decide that these new requirements need to be incorporated in the product line assets, several product line assets may be affected. Changes to a shared asset will affect all products that incorporate the asset and new requirements need to be integrated carefully. One risk is that the extensions to, e.g., a software component are frequently too product-specific and are not sufficiently generalized. In figure 1, the evolution process, fuelled by the need to incorporate new requirements, is visualized.

We have identified a number of categories of requirement evolution. Below, we present an introduction to these categories and briefly discuss the effect changes in each category can have on the product family architecture and the product family components. In [2], the categories are discussed in more detail and exemplified.

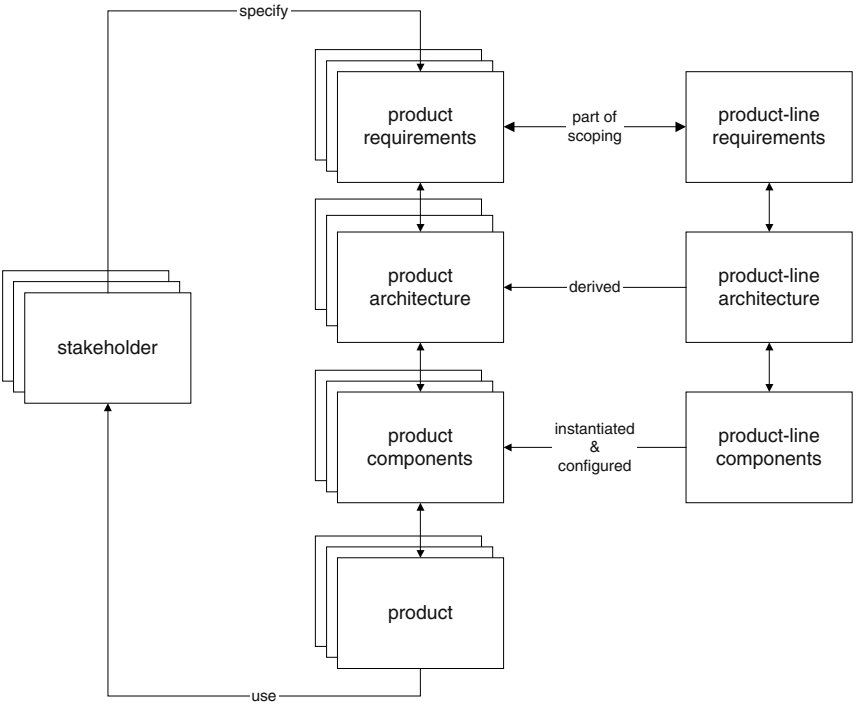


Fig. 1. Evolution in software product families

New Product Family

The first form of evolution that we discuss in this chapter is also the most extensive one, i.e. the introduction of a new product family based on an existing product family. This situation can occur due to a number of different circumstances. Below, we present a number of possible causes for initiating a new product family.

- **Too large variability**: A company may see an opportunity to release a set of software products on a new market or market segment based on its existing product family. However, the set of new products is different in at least one major aspect of the product behavior. For instance, different hardware, operating system or user interface mechanisms.
- **Business reasons**: Especially when products in the product family are incorporated into larger systems by the customers of the organization managing the software product family, business opportunities may occur that require the organization to initiate a new product family based on the existing one.
- **Incorporating independent products**: Either through merging with or acquisition of other companies in the same market segment or by extending the scope of the software product family, it may become necessary to incorporate independent products into the product family. Since the products that are to be incorporated generally do not share the same architecture and components, converting these products at once may be very costly and not economically viable. In addition, these products may exhibit relevant features that currently are not included in the product family.
- **Geographical distance**: A fourth possible reason for instantiating a new software product family is the geographical distance between the units maintaining the product family and the units maintaining a subset of the products in the product family. Due to the geographical distance, and associated communication problems resulting from, among others, different time zones, etc., it may be too complicated to maintain a highly integrated product family. A solution to this approach may then be to create a new product family within the existing product family.
- **Cultural conflicts**: In some cases, the introduction of a software product family causes substantial cultural conflicts to surface between the various organizational units that were affected by the new product family. However, even when a product family is well established, problems of non-technical nature may appear that require management to reconsider the product family approach. Often, due to reorganizations the organizational units working with product family assets may obtain increasing levels of independence and, on occasion, the product family may restrict the agility of the individual units. In those cases, tension between that unit and the others will surface and forces to increase independence by leaving the product family and maintain product-specific versions of all but a few components.

The above reasons may require the introduction of a new software product family. One can identify two approaches that can be used to establish the new product family, i.e. cloning or specialization. Below, these approaches are discussed in more detail.

- **Cloning**: In cases where local control and agility are the primary goals, the preferable approach is to clone the existing product family, i.e. literally create an additional copy of all assets that are part of the product family. The organizational unit or units that demanded their own product family become responsible for the new copy of the product family. Generally, the first step is a pruning activity in

which all components and architectural elements in the product family that are not necessary for the products supported by the involved unit or units are excluded from the new product family. Subsequently, often a redesign of the product family architecture and an evolution of the software components to remove irrelevant requirements and to incorporate the new requirements is required.

- **Specialization**: The second, less revolutionary, approach is to specialize the product family. Specialization, in this context, results in the creation of a new product family within the existing software product family. Analogous to object-oriented programming, the inheriting product family obtains all features and assets in the parent product family, but is able to override the existing features and assets and may extend the inherited features and assets with new functionality. If one understands that the products in the product family already exhibit a specialization relation with the product family, it is clear that a specialized product family typically is inserted in between the existing product family and a subset of the existing or new products. The specialization approach to creating new product families is a highly viable way of working that maintains many of the advantages of a product family approach.

Introduction of New Product

The second type of evolution, less demanding fortunately, is the introduction of a new product in the product family. The addition of a product to the product family can originate from a number of different causes. Below, some of those causes are discussed in more detail.

- **Market opportunity**: The first, and most frequent, reason for adding a new product to the product family is the identification of a market opportunity. The organization has identified that a certain segment in the market is not serviced by the existing product family and that there is a business case for entering the segment.
- **Incorporate independent product**: A second scenario that leads to the extension of the product family with a product is when an independent product needs to be incorporated. One cause for adding a product is when the original decision with respect to scoping is changed by management. Management decides that a certain product should be incorporated in the product family, even if the initial decision was against incorporating the product. A second cause may be the acquisition of a company that develops a product that falls within the scope of the product family.
- **Extend product family scope**: Finally, a typical scheme to adding new products to the product family is by extending the product family with products that are at the high-end and at the low-end of the products currently supported by the product family. The process often starts with the identification that the product family is able to support an additional product family member, if only a particular feature or feature set is added to the product family.

Especially this last type of adding products to the product family is frequently complicated by conflicting quality attributes. The current product family is optimized for a particular set of quality attributes that is the result of a compromise. Since the compromise often is optimal for the products in the middle of the product family, adding products at the extremes, be it high-end or low-end, often requires that the

software architects and engineers explicitly address the conflicts between the quality attributes in the product family and the quality requirements of the new product.

To incorporate a new product, a number of steps have to be performed. Below, these steps are discussed in more detail:

- **Identify commonalities**: The first step must be to identify the commonalities between the product and product family. Since the decision has been taken to integrate the product in the product family, there must be considerable overlap between the two entities. Thus, using the feature set defined during the design of the product family architecture, one can check for each feature whether the product implements or requires that feature as well. In the case where the feature is currently not incorporated in the product, it should be considered whether including the feature will have any negative effects for the product. If incorporating a feature in the product does not have any negative effects, but would require less variability from the components implemented in the product family, it may be beneficial to incorporate the feature.
- **Match product and product family architectures**: Once the functionality required by the product has been identified, the software architecture for the product can be matched with the product family architecture. Especially when integrating an existing product, a compatible software architecture will simplify the integration process. In fact, if the software architectures are very dissimilar, it may be impossible to integrate the product. If the architectures are structurally similar, replacing product-specific components with product family components can generally be performed at a relatively small effort.
- **Develop variation points for product family components**: In general, an independent product that is incorporated in the product family will have variability and functionality requirements on the product family components that are currently not supported. The components need to be extended with support for the new product as well. This can be achieved by extending the product with additional variation points, that allow for the appropriate configuration of the component. If the product-specific requirements can be achieved using this approach, that is preferable. Otherwise, other approaches are required, as discussed below.
- **Develop/reengineer product-specific code**: Where the product family assets do not facilitate the product requirements, it is necessary to implement product-specific code. This can be implemented by developing product specific extensions to product family components that support this, e.g. object-oriented frameworks. Alternatively, product-specific components should be developed that support the missing requirements.

  In the case of an existing product being integrated in the product family, the functionality not provided by the product family is present in the code of the product. Assuming up to date documentation is available, the relevant parts of code need to be separated from the remaining product code. Otherwise, the existing code may need to be reverse engineered in order to document its structure. Subsequently, the parts have to be integrated into a software component and the provided, required and configuration interfaces should be specified. Finally, the component should be verified against the requirements.
- **Instantiate the product**: As last step, the product should be instantiated based on the product family assets and verified against the requirement specification for the product.

Adding New Features

The third type of product family evolution, again less dramatic than the previous type of evolution, is the inclusion of new features in the product family assets. Typically, there exists an upward pressure in a product family in terms of the functionality in that features often are implemented first as product-specific code for a single product, then it is generalized for the most specialized product family in the hierarchy. This upward movement continues until the feature has reached the top-level product family.

Adding new features is, in our experience, the absolutely most common type of extension to the product family. The new features can originate from a wide variety of sources. Below, some typical types of evolution are discussed:

- **Market investigations**: Either passively or proactively through active research by the marketing department, demands for new features available for the existing products generally appear constantly. These new features cover a wide spectrum. However, these features have in common that they originate from the perspective of the customer, i.e. the features satisfy an existing market, rather than creating a new one.
- **New technological opportunities**: With the advancement of technology, new possibilities arise for products that were not available earlier. These possibilities should be exploited, but since there generally is not yet a market available for these new features, effort must be put on creating the market as well as developing products that exploit the new features.
- **Competitors**: Finally, a company can take the strategic decision to support a new feature throughout its product family because its competitors provide, or are expected to provide the particular feature. These features may neither be requested by its customers nor the result of technological progress, but may be necessary for maintaining or expanding market share.

The incorporation of new features in the product family is similar to traditional maintenance activities. However, one difference is that new features may not be relevant for all products in the product family, but only for a subset of the products. How such features should be incorporated is an issue that needs to be addressed explicitly by the architects and engineers working with the software product family. One of the potential problems is that extensions required to incorporate the new features may have negative effects on the products for which incorporating the new features is not relevant.

In each situation where new features are positive for one subset in the product family, but negative for the remaining products, the possibility of creating a hierarchical product family should be considered. That allows for incorporating the conflicting features in the products that need them while avoiding the negative effects for the remaining products. However, this is only feasible for features that are relatively modular in nature. For instance, quality requirements that have product family-wide effects cannot be incorporated in this way without cloning the product family. The negative effects of the latter often outweigh other concerns and should be avoided where possible.

Extend Standards Support

In the first version of a software product, standards are often not implemented completely, but only the part that is required for the correct operation of the product in normal modes. In addition, standards, although the name indicates stability, also evolve through the frequent release of new versions. This leads to the fourth type of product family evolution, i.e. the extension of standard support by the products in the product family. In our cooperation with companies, we have identified a number of types of standards that typically are implemented in an evolutionary fashion. Below, we discuss three types of standards in more detail.

- **Network communication protocols**: These types of protocols are a typical example of standards that are extensive, evolve constantly and where the 80/20 rule applies, i.e. 80% of the functionality defined in the protocol can be achieved by implementing 20% the standard.
- **Component communication standards**: Component communication protocols such as Microsoft's COM, Sun Microsystems' JavaBeans and, especially, OMG's Corba specify challenging amounts of details about the correct operation within their respective standard. However, for being able to operate as a component using the standard, in most cases only a limited subset of the standard is required. This allows for partial implementation in the first releases and the gradual extension of the support for the component communication standard(s).
- **File systems**: Even file systems, especially networked file systems, exhibit the property where only the core part of the functionality needs to be implemented for the product to be useable in a file system context. For instance, in a networked file system, it is possible to not support references to other nodes. In that case, product can only be used as a leaf in the network, but for many types of products that may, at least initially, be satisfactory. For more advanced features, the file system implementation will provide interfaces that either return error codes or other types of dummy replies.

Incorporating additional aspects of a standard is similar to the incorporation of new features, which was discussed in the previous section. We refer to the earlier sections for discussions on how to incorporate the new functionality. One difference is that standard support generally is less product specific than features, although product families exist in which the products distinguish themselves based on the supported standards. In addition, standards are generally well modularized and do not cause product-wide effects because of quality requirements. Consequently, there will be less tension between the products in the product family when incorporating extended standard support.

New Version of Infrastructure

The types of evolution that we have discussed up to now are primarily concerned with the incorporation of new functionality in the product family. The type of evolution discussed in this section is concerned with the opposite. In a number of cases, we have identified the situation where a new version of the infrastructure, e.g. hardware and operating systems, underlying the products in the product family implements functionality or behavior that, up to that point, was implemented as part of the product

family. In certain cases, it may be possible to ignore this fact and to continue to use the functionality as provided in the product family assets. However, the disadvantage is that those maintaining the product family assets remain responsible for the evolution of this functionality. In addition, the implementation provided in the infrastructure is often complete and up to date, whereas this is not necessarily the case for the implementation that is part of the product family.

Therefore, assuming the new version of the infrastructure incorporates relevant functionality, it is, in most cases and especially in the long term, preferable to make use of the functionality provided by the infrastructure and to remove the now redundant functionality from the product family. However, the disadvantage is that a, potentially substantial, amount of effort is required to incorporate the changes. Different from the earlier types of evolution where functionality was added to the product family, the focus of the maintenance work is in this case not in the component providing the functionality since this component is simply removed from the product family asset base. Rather, all the clients of the component need to be changed in order to invoke the infrastructure interfaces rather than the original component.

Changes that affect all clients of a particular component illustrate the importance of separating the binding of required interfaces from the component specifying the interface. If the product family architects have achieved a complete separation, then, in the simplest case, the only required change is to 'rewire' the required interfaces of the client components. However, in practice the interface provided by the infrastructure is not identical with the interface provided by the original component. In that case, either a proxy component can be introduced that converts the calls to match the infrastructure interface or all clients need to be changed in order to invoke the infrastructure appropriately.

The most complicated case is where the functionality provided by the new version of the infrastructure is not equivalent with a single component in the product, but is only part of a component or, even worse, is distributed over multiple components. In that case, the effort required to incorporate the additional infrastructure functionality is often substantial and most organizations will choose to proceed in an evolutionary manner, i.e. incorporating the functionality over the course of a number of iterations.

Finally, we discuss three categories of infrastructure that may improve and affect the product family:

- **Hardware**: Among others due to Moore's law, one can recognize a constant evolution of the functionality provided by hardware devices, such as microprocessors, communication devices and application-specific devices such as ASICS. In product families that do not depend on an operating system, but implement concurrency, new features for context switching provided in the microprocessor hardware will affect the handling of concurrency in the product family. In addition, in communication-oriented processors, the lower levels of communication protocols may be implemented in hardware, freeing the product family functionality from that task.
- **Operating system**: Operating systems such as Microsoft Windows and the various Unix variants, such as Linux, are evolving to become more and more component-based systems, i.e. the various parts in the operating system, such as memory management, thread and process management, device management, etc. are implemented as components rather than a single monolithic entity. The set of components that is considered to be part of the operating system is constantly

increasing. This development has effects on virtually all product families that run on a commercial operating system.

- **Third-party component**: Our third example may not indicate infrastructure in the traditional sense of the word, but many product families incorporate a number of external components that provide important functionality for the product family members. An example is a web-server component that allows software products to be accessed through a web browser. Several freeware and commercial web-server components are available that can be incorporated in the product family. However, these web server components are constantly improving the supported functionality and easily touch on functionality that traditionally, for instance, was considered to be database management (DBMS) functionality, such as support for various types of querying mechanisms.

It should be noted that all elements in the context of the software product family, be it hardware, operating systems or third-party components, may change in undesirable ways that may strongly affect the products in the product family. Although the use of externally developed elements is generally preferable from an cost/benefit perspective, one should be aware of the increased risk level due to reduced scope of control.

Improvement of Quality Attribute

The final type of evolution that we discuss here is not concerned with the functionality provided by the product family, but rather with the quality attributes of the product family members and of the assets in the product family. Typical examples are the demand for improved run-time quality attributes, such as performance and reliability, and design-time quality attributes, such as flexibility and variability. A typical scenario is that the developing organization is eager to minimize the time-to-market (or delivery) for the first version of the product and in the process, attributes such as the aforementioned are sacrificed. During later versions, these attributes need to be improved in order to satisfy the users of the product, that generally complain about the, less than satisfactory, product properties.

Quality requirements frequently have product-wide effects on the structure of the product. Consequently, in order to improve the quality attributes, changes with architectural impact may be necessary. However, at this point we have reached the stage where changes to the product family architecture will have effects on the software components and products that are part of the product family, i.e. architecture transformations are immensely more expensive than during the initial design of the software architecture for the product family.

However, although many cases exist where architectural changes cannot be avoided, there are several techniques available that are more local in their nature and that may improve the relevant quality attributes at least with a reasonable amount against a fraction of the cost of architecture redesign. Below, we discuss some examples of such techniques:

- **Cache**: One technique that has been used in a variety of situations is caching. By storing information in a redundant but quickly accessible form, the average delay when accessing information can be drastically decreased. Caches are typically used in microprocessors and as part of disc drivers, but also web servers and a large

variety of other applications have improved performance using a cache. The main disadvantage of using a cache is that it requires additional resources, generally memory, to improve the utilization of another resource, generally I/O bandwidth or CPU cycles. A second disadvantage is that, depending of the situation, e.g. read-only or read-write access to the information, the redundant representation can lead to inconsistencies. To avoid this, it is generally required that all access to the information is performed via the cache.

- **Memory management**: A number of authors, e.g. [2], have indicated that especially in concurrent object-oriented applications running on parallel hardware, considerable amounts of the total computation (up to half) is spent on memory management, i.e. the allocation and deallocation of objects. The introduction of an object pool where objects are returned after use and can be retrieved when needed has shown to increase performance drastically in those type of applications. In general, investigating the fundamental cause of performance bottlenecks and replacing some general-purpose functionality in the infrastructure with an application-specific solution can have enormous improvements in performance as a result.
- **Indirect calls**: One approach to increasing, especially run-time, flexibility is to insert a level of indirection in the communication between the components in the product. By inserting such indirection, invocations can dynamically be redirected, which improves the possibilities for dynamic replacement of components, synchronization of behavior and the insertion of additional behavior where necessary. The disadvantage of this approach is, obviously, run-time overhead and additional initialization code.
- **Wrapper**: The incorporation of a quality attribute may require several components to change their behavior, which is an expensive operation to perform. In those cases where the functionality can be added before or after an operation in the component is performed, e.g. encryption and decryption, the concept of a wrapper can be a useful mechanism to achieve this. Especially when multiple components need to be extended with identical behavior, a wrapper is effective from a development perspective, since one instance of the wrapper can be created for each component. The primary disadvantage of using wrappers is that there often is a measurable associated performance penalty. For instance, [1] identifies that wrapping may lead to large amounts of glue code and serious performance degradation.

Change and Evolution of Software Family Architecture

It has become a cliché that change is the only permanent factor of software development. We also commonly regard it as a necessary evil.

But should change in the architecture be taken in the same way or should we regard it in a different manner? Does change in the architecture always mean we overlooked something? Does change in the architecture in the late implementation phase always mean bad news? Is any kind of change and variation in the architecture of software family acceptable? Does a change in the choice or interconnection of components in a variant product imply a change in the family architecture?

In this section we present some views on these questions. These views should not be regarded as final in any respect. Their main purpose is to evoke discussion of these topics and attention to the subject.

Architecture Change May Be a Good Sign

Sometimes a change in architecture descriptions does not reflect a change of the architecture, but rather a change in our understanding of the system, although some may argue that our understanding IS the architecture. In any case such a change should not be regarded as evil, but rather as a positive development, maturation and evolution of the architecture in the true sense.

For example in one of the projects a few years ago we faced a peculiar situation. We were building a network product that supported persistent configuration data for line cards without central storage. That means that a newly inserted line card was "by default" configured using last good configuration of the card in the same slot. Configuration information had to be at all times replicated to multiple cards so it could survive removal of an arbitrary card. Naturally, this functionality was hidden inside card default configuration component and did not show in the top-level architecture diagrams.

Well into the implementation we realized that much of the functionality of the default configuration component was about implementing persistent storage on a node that did not have any "central unit". The implementation was clean though and properly layered. We decided there should be an explicit component in the architecture that would manage persistent data. This would be useful in a fully distributed node for other components that needed persistent data. This also would be useful when implementing a node with a central unit that can host persistent data. Since much of complex functionality of data replication will be kept separate from other functions it can be removed when not needed.

Project management did not permit the change in the architectural description of the system. The argument was that a change in the architecture and introduction of a new component so late in the project might compromise implementation schedules. In reality the change would enable several groups to better utilize already developed functionality and would better support variation and would not require any change, except in the interpretation of what we were building.

Architecture Change May Be a Bad Sign

We have learnt to accept change in software development and, especially when we speak of product families, change and variation seem to be the common case rather than an exception. One should however keep in mind that proper architecture should prevent unnecessary change.

Here is an explanation by example. An interworking unit is situated between two different networks. Its function is to translate signaling and data of one network into those of the other. Say, we are building a family of interworking units (a job that needs to be defined in a lot more detail of course). In a typical situation there would be a (at least one) specific kind of line card for each kind of network that our unit can

interwork. The line cards are physically interconnected with some kind of cross-connect or bus technology. See **Fig. 2.** 2.

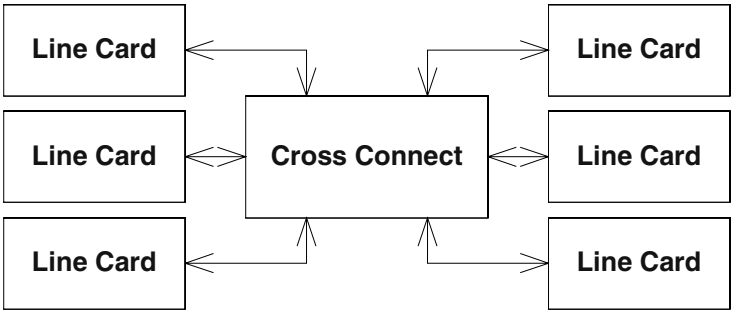


Fig. 2. Physical structure of an interworking unit

Fig. 3. is a simplified illustration of an almost realistic architecture of data plane software for a line card of an interworking unit. The incoming traffic is analyzed and the line card to which it should be directed is determined by the internal routing component. It is then sent over some cross-connect device to the appropriate line cards. There it needs to be converted so it can be sent out to the other network.

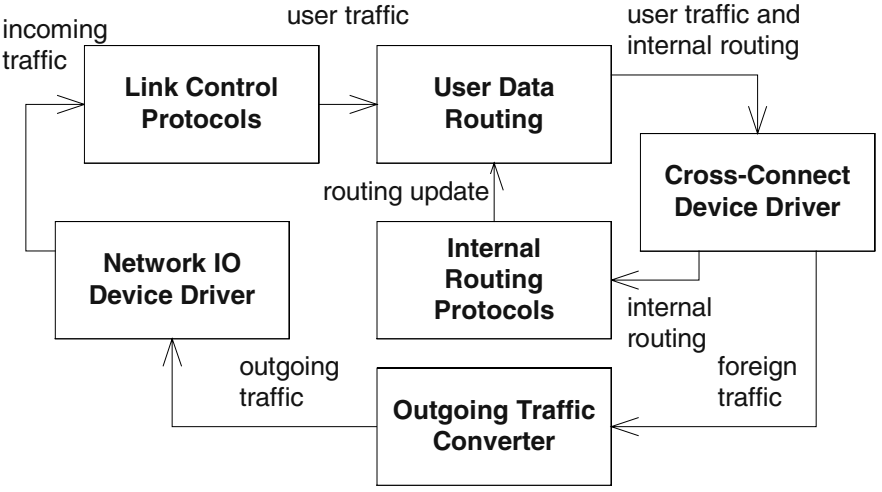


Fig. 3. A bad example for architecture of line card software

The software on the line card is specific to the network it is connected to. Since it needs to accept traffic from cards that are connected to other networks however it also depends on the kinds of the networks the other line cards may be connected to. As a result every time there is another kind of network we want to support with our interworking units, the outgoing traffic converter component for every supported network should get another variant. If we are planning to interconnect N networks we

will need N^2 of outgoing traffic converter components. Although internal routing of incoming traffic should not depend on the specific network the line card is connected to, in this design, it seems we will have to maintain a variant per network.

There is, of course, a well-known solution that avoids these problems that uses an internal representation and only requires converters from and to this internal representation. This is illustrated on Figure 4. If we are planning to interconnect N networks we will only need to develop and maintain 2N of traffic converter components. Also common functionality such as internal traffic routing will be part of the family architecture and will be reused without change with all line cards.

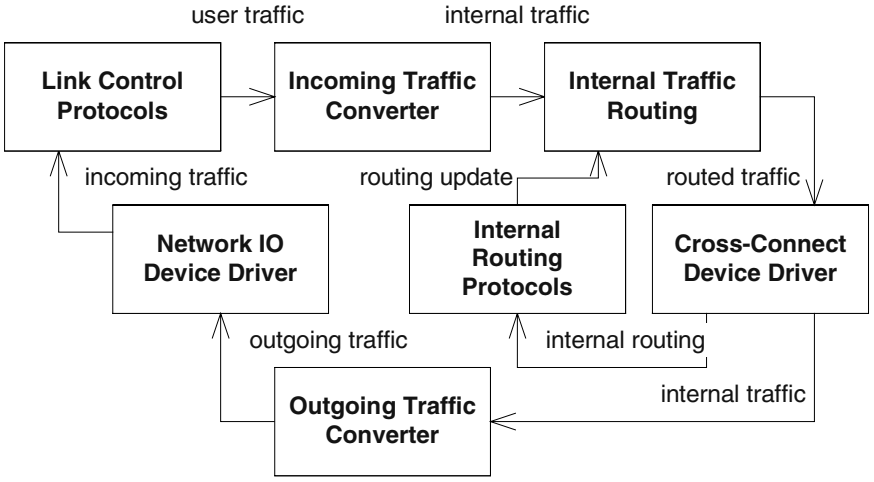


Fig. 4. A better example for architecture of line card software

Architecture Change May Be a Misunderstanding

Architecture is always an abstraction. One implication of this fact is that architecture of a single system does in fact describe a class of systems with potentially significant variation in their function, features, and quality attributes. How then architecture of software family is different from architecture of a specific system?

This is not a philosophical question; at least not only. We think that much of the perceived change in the architecture of software families is due to misunderstanding of what constitutes family architecture in the first place.

For one, when talking about architecture of software family we may need to revise our understanding of software architecture being the interconnection of main components. In most software families the choice of components and their interconnection may change significantly in each variant product. If there is something that can be called family architecture it probably needs to be concerned with something else.

To communicate a concept one can define it or refer to a prototypical instance. Our current approach to family architecture is that of prototypes. Family architecture is represented by the architecture of a prototypical system. The prototype approach has

well-known limitations: abstract concepts have no prototypes, prototypes over specify the concepts they represent. For example, the prototype may have specific properties that vary significantly among the instances of the concept.

The same is true when the prototypical approach is used to represent software family architecture. Therefor much of the change we perceive in family architecture is not an essential change. If we could separate the definition of the family architecture from accidental properties of the prototype, we would perceive less change in the family architecture and would also be able to better preserve it against unintentional alteration.

Architecture can be described as a set of decisions or assertions about the structure and the texture of software. What we propose as the first step is to group the assertions based on the sets of systems for which they are true. Each set of systems forms an architectural scope. Interesting groupings of architectural decisions partition the corresponding system instances into a set and its proper subset, or non intersecting sets.

Subsets correspond to more specific architectural scopes as for example product family within an application domain. Non intersecting sets of instances represent architectural scopes of variants as for example variant products within a product family, or product families in an application domain.

This is of course just a way to think about it. In reality nothing fits clean, simple classifications, or makes tree structures. Still some progress can be made by keeping in mind an ideal. A useful hierarchy of architectural scopes may include

- variant architecture,
- dynamic variant architecture,
- evolving system software architecture,
- family software architecture,
- domain-specific software architecture,
- reference architecture.

Let us shortly describe characteristic concerns of few of these architectural scopes. Domain-specific software architecture (DSSA) defines essential domain concepts and functional partitions (structure and APIs) that enable development of shared platforms, components and component frameworks for construction of software in the specified domain. DSSA is typically concerned with domain-specific infrastructure. This may include selection of operating system, middleware, and data persistency services. DSSA may also specify other components and component frameworks providing more domain-specific functionality that is useful for most systems in the domain. For example DSSA for communication network elements may include a node configuration framework, user authentication and access authorization components. DSSA should be defined on the level of abstraction on which the variation between different systems in the domain is not visible.

Software architecture of a product family (SAPF) defines concepts, structure and texture necessary to achieve variation in features of variant products while achieving maximum sharing of parts in the implementation. It is similar in intent to DSSA except that while focus of DSSA on commonality and uniformity the focus of family architecture is on achieving variability. An appropriate scope for the family can be determined by bounded complexity of family architecture. When there are too many differences between variant products or too many dependencies this will be reflected in the complexity of the family architecture. This is why SAPF defines a smaller

architectural scope than DSSA. Note that product families exist primarily on the implementation level and therefor may include variant products intended for different use. Those variant products may differ from each other in their essential features. This often implies that SAPF does not make decisions regarding specific product component structure, but only establishes patterns or mechanisms for creating structures common in the product family

Evolving system software architecture (ESSA) defines the stable structure and flexibility parameters of the specific system. ESSA defines support for variability in the capacity of the essential features and selection of the secondary (or optional) features provided by the product. ESSA specifies the component structure of the system in write-time and run-time component domains. ESSA is specifically focused on provisions for extension and evolution as well as variation in optional features and feature capacity as may appear in a single product evolution or in a line of products that coexist at the same time.

It is essential that architectural descriptions will be properly partitioned along this hierarchy of architectural descriptions. Thus if the target is creation of an evolving system the specific variant architecture needs to be described in relation to dynamic variant architecture, which is in turn defined in relation to evolving system architecture.

The paper "A Two-Part Architectural Model as Basis for Frequency Converter Product Families" presented in this workshop by Hans Peter Jepsen and Flemming Nielsen is a fine example of how architecture needs to be to defined for different architectural scopes, and how architecture defined in more general architectural scope creates the context in which architecture of the more specific scope is defined. They partitioned architecture of frequency converter control software to three architectural scopes. The most general scope grouped decisions that are applicable to a wide application domain of input-enabled control systems. These were mainly expressed as separation of continuos processing done in "control loop" and event based input of control parameters. The next most general architectural scope collected decisions regarding the communication mechanism employed between event driven component cluster and components in the control loop. The third architectural scope explored in the paper has grouped decisions regarding specific functions performed by frequency converters.

It could be worthwhile in this example to further separate and group architectural decisions to better control change and evolution of architecture. It, of course, always depends on the specific case which architectural scopes should be separated and explored independently from the others.

Discussion

The discussion at the evolution session touched upon several interesting points. Henk Obbink emphasized the fact that however well we plan the architecture to be future-proof the future will always surprise us. Changes to architecture are inevitable and understanding how to introduce the changes and when it is worth while to integrate old variant products with family architecture that has evolved is non trivial.

David Weiss raised an important question: What are the resources that that allocated to manage evolution of product families at different companies. This is a

single number that can indicate to us the perceived importance of architecture evolution in software development. We were not able to get any coherent answer or estimation during the session, but the question deserves close attention by researchers in the field.

Jean-Marc DeBaud introduced the topic of economic viability of software family evolution. He pointed out that there was little mention of criteria upon which the decisions to evolve or not to evolve an architecture could be made. In particular, when is it beneficial business-wise. This should be one of the primary issues to discuss, though of course, the technical evaluation of an architecture in the light of a proposed change is principal as well. This leads back to the notion of economies of scope. As Jean-Marc put it: "To me the evolution of any family platform can only be relevant, together with possible added new business rationales. If the evolution take place within the constraints placed on the family by the originally set (or later modified) economies of scope, then it makes sense. If the proposed evolution takes the platform outside those economic boundaries, then that evolution should not be done. As we have put together a method to define the economy .of scope and truly propagate them at the architecture level, I think we are building a framework to put a family engineering without too much over- or under-design of the core assets."

References

[1] U. Hölzle, 'Integrating Independently-Developed Components in Object-Oriented Languages,' Proceedings ECOOP'93, pp. 36-56, 1993.

[2] Daniel Häggander and Lars Lundberg 'Optimizing Dynamic Memory Management in a Multithreaded Application Executing on a Multiprocessor', Proceedings of ICPP 98, the 27th International Conference on Parallel Processing, Minneapolis, USA, August 1998.

[3] Mikael Svahnberg, Jan Bosch, 'Evolution in Software Product Lines: Two Cases', *Journal of Software Maintenance*, Vol. 11, No. 6, pp. 391-422, 1999

Product Family Techniques Session

David M. Weiss

Lucent Technologies Bell Laboratories
Naperville IL 60563, USA
weiss@research.bell-labs.com

The papers in this session were mostly based on the authors' experiences in creating industrial product families, describing the difficulties they had, and the solutions they found to them. Most started with some set of techniques in mind and modified them as necessary to make them more workable in real situations. The following papers were included in this session.

- Beyond Product Families.
  Context: Several consumer electronic product lines, with commonality across them
  Problem: How to take advantage of sub-families across product lines?
  Solution
  - Subsystems were used as the unit of commonality.
  - Interfaces were strictly controlled, with variability and commonality controlled through the interfaces, using a module interface language for explicit interface specifications
  - A disciplined approach was taken both to the development process and to control of the product population architecture.

  The techniques described have been and are being used in creating the members of a product population in the consumer electronics industry.
- Requirements Modeling for Families
  Context: Family of safety-critical medical systems with much customization/many potential configurations
  Problem: How to agree on and control requirements and specify them precisely?
  Solution:
  - A family of requirements documents designed for different audiences and purposes
  - Features, use cases
  - Requirements object model for concepts, UML-based
  - Provide intellectual coherence

  The emphasis was on making the concepts underlying the family precise and controllable. The techniques have been and are being used in creating a family of medical imaging systems.
- Creating Product Line Architectures
  Context: The scope of a product line has been defined, and commonalities and variabilities have been analyzed, using the PuLSE process.
  Problem: How to create a family architecture systematically?
  Solution: Use an iterative process based on scenarios (use cases)
  - Group scenarios by importance
  - Create evaluation plan (test cases)

F. van der Linden (Ed.): IW-SAPF-3, LNCS 1951, pp. 184-186, 2000.

•Derive architecture (components, relations, decision model, mechanisms)
•Evaluate

This process is in the stage where it is ready to be tried on a real industrial problem.

- Extending Commonality Analysis
  Context: Embedded systems with considerable environmental interaction and safety-critical, real-time concerns, particularly aircraft engine thrust-reverser systems
  Problem: How to deal with interactions in commonality analysis, and with varying detail in information
  Solution: Introduce hierarchical structuring among commonalities and variabilities resulting in a layered structure, where each layer is defined by a set of variabilities with an accompanying set of commonalities determined by fixing the decision for the corresponding variabilities.
  The method provides both conceptual and notational support for dealing with the layered structure.
  The technique has been applied to a family of aircraft engine thrust-reverser systems.
- Stakeholder-assessment Of Product Family Architecture
  Context: Product line architectures for information systems are mostly derived from existing systems
  Problem: How to evaluate family architectures with involvement of "family-naive" stakeholders?
  Solution: Extend SAAM
  •Plug-in evaluation modules for family (and other) concerns
  •Includes both process steps and assessment methods
  •Add assessment definition and conformance steps, with family planning in mind

Several themes were clearly apparent in all of the papers. All focused on the need to be explicit, to be precise, to be systematic, and to manage complexity. These themes were apparent in the use of techniques for controlling concepts and architectures, for documenting the results, and for introducing disciplined processes that maintained conceptual integrity.

During discussion of the papers, the following issues emerged.

- How to prevent design/implementation details from creeping into requirements. This, of course, is an issue that appears in all requirements determination methods, not just those for families. Design or implementation details that appear in product family architectures may result in an unintended distortion of the scope of the family, allowing unwanted family members to appear, or desired family members to be difficult to produce.
- What's the difference between internally developed components, e.g., subsystems, and externally acquired components? Externally acquired components may be less expensive, but may be more difficult to use for the particular family, or may be more difficult to maintain over time if they are not appropriately documented. The vendor may not be responsive to the needs of the product line for the buyer. Internally developed components may be expensive to maintain and may not take advantage of new technological developments that appear in the marketplace.

- Can properties such as safety be reasoned about in a family context similarly to system context? This is a difficult problem for which there is not yet a good solution.
- Does complexity added by family documentation make architects disinclined to use it?
- What are limitations on testing family architecture, esp. if you can "only" build prototypes? When is it safe to build a prototype?
- What verification can you do without prototypes, e.g., consistency checking on specification?
- In the product population approach, there is no need to integrate subsystems into platforms in order to build a system. This allows subsystem independence in delivering versions.
- The need for external reviews of architectures is perhaps more critical for family architectures than for architectures for single products, since there is more at stake in a product line architecture, and since the success of a product line architecture critically depends on the ability to forecast the future needs of the family.

Nearly all of the papers showed an admirable tendency to reason from and learn from experience, rather than to speculate based on theory.

Beyond Product Families: Building a Product Population?

Rob van Ommering

Philips Research Laboratories
Prof. Holstlaan 4, 5656 AA Eindhoven, The Netherlands
Rob.van.Ommering@philips.com

Abstract. Building a large variety of products in a global organization, with software development distributed around the world, requires an approach which must not only have a sound *technical* basis for handling diversity and commonality, but where also the software development *process* and *organization* must be aligned optimally. In our case, the diversity of products is so large that we'd rather speak of a product *population* than of a product *family*. We find it helpful to use an approach that emphasizes composition over decomposition, and that embodies different types of processes (architecture, subsystem and product development) that are mapped to development sites in the organization.

Introduction

Philips Consumer Electronics (CE) produces a large variety of products with embedded software, for example televisions, set-top boxes, video recorders and DVD players (see Figure 1). These products are managed as different product families, where each family has variations due to differences in e.g. standards, world region and price setting. The Philips CE organization is divided into business groups, where each group is responsible for one of the product families mentioned above.

Fig. 1. Example products.

Philips Semiconductors (PS) is the main supplier of many of the ICs used in CE products, and – reversibly - CE is an important (internal) customer for PS. But PS also has other (external) customers, and PS not only delivers the ICs, but also software with the ICs. Sometimes even, PS provides partial or full hardware solutions (printed circuit boards or complete chassis), together with the software controlling that hardware. PS is also divided into business groups, where each group manages a family of hardware/software products.

F. van der Linden (Ed.): IW-SAPF-3, LNCS 1951, pp. 187-198, 2000.

Here is our problem. Using traditional approaches, such as applying software libraries or object oriented frameworks, variation within single product families as mentioned above can be managed. But for us, there are two (business) arguments why this alone is not sufficient:

- The 'weak' argument: there is also a lot of technical commonality *between* the product families, so duplication of effort can be avoided. The argument is weak since the problem can in principle be solved by hiring more people.
- The 'strong' argument: we are missing the opportunity to sell *combinations* of products, such as TV-VCR combi's and TVs with integrated set-top boxes. We have the parts readily available, but since they are realized in different business groups as different product families, they cannot easily be combined.

Here is our challenge. We must define a *product population* approach that allows our business groups to optimally use each other's software, while at the same time leaving sufficient freedom to realize their own kinds of products. This not only requires sound technical solutions, but it also affects the development process and organization. Note that we distinguish between a *single product* (television), a *product family* (televisions) and a *product population* (CE products, see Figure 2). The large variety of products in the latter forces us to make fewer assumptions at the global level, and to rely more on composition than on decomposition.

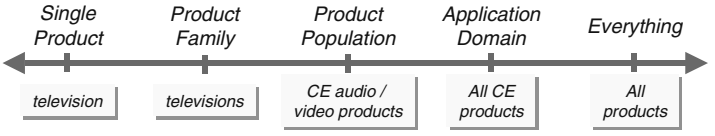


Fig. 2. The spectrum of product diversity

The paper is organized as follows. After listing our most important sources of inspiration in the next subsection, we describe our technical solution, our architectural solution, and our solution for process and organization. We end with concluding remarks.

Sources of Inspiration

Our most important sources of inspiration are listed below.

- The *Reuse approach* as advocated in [1] makes a distinction between application family engineering (AFE), component system engineering (CSE) and application system engineering (ASE). Basically, it separates the creation of reusable *components* (CSE) from the creation of *products* (ASE), where both take place under a common *architecture* (AFE).
- The *Semiconductor business* already builds and sells reusable components for years, in the form of devices such as transistors and ICs, but also as printed circuit boards and (recently) reusable *core cells* for creating ICs. The electronic hardware domain has a long tradition in the creation of for instance interface and component data sheets and of component catalogues.

- *Microsoft COM* [2] allows software components to be developed independently from each other, and to be used in a variety of products, one of the necessary preconditions for a component industry. It is not as much COM's *location and language transparency* that we value (although these will be very important for us in the near future), but more the clear *separation between interfaces and components*, the *immutability* of interfaces, and the ability to cope with the *evolution* of components.
- *Darwin* [3] is an architectural description language that supports a component oriented approach with an *explicit* (instead of implicit) architecture, which helps to manage the *complexity* of products and components. Darwin also offers a mechanism to make *context dependencies* of components explicit in the form of requires interfaces.

The Technical Solution

We create product populations with a *compositional approach* rather than using for instance a classical decomposition enriched with variation points [1]. Based on an extensive knowledge of the full domain, we identify basic *software components*, which we can combine in different ways into *standard designs* or *subsystems*, which we can combine again in yet different ways into *products* (see Figure 3). More levels of aggregation may be defined where necessary. As product population architects, we have to manage the full composition graph.

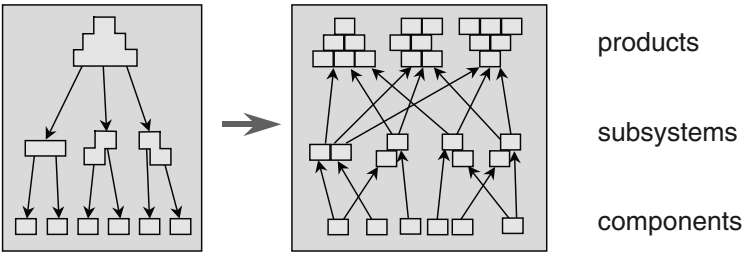


Fig. 3. From decomposition to composition

This allows us to create products that are quite different from each other, while still sharing non-trivial pieces of functionality. To achieve this, we must have:

- a *component model* that allows us to define components and use them recursively to build new components until products are obtained.
- a way of making the *architecture explicit*, to *manage* the composition graph.

The component model that we currently deploy is called Koala. It inherits from COM, and it contains an architectural description language based on Darwin to make the architecture explicit. Koala is described in more detail elsewhere [4]. Here, we highlight only a few of its distinguishing features.

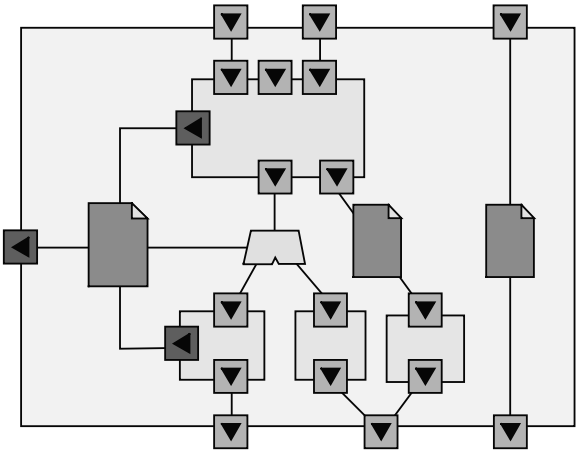


Fig. 4. An example Koala component

Components

A Koala component is an encapsulated piece of software where *all* context dependencies are routed through interfaces to be connected by the 'user' of the component. This user can be a 'compound' component (see Figure 4), which makes the model recursive. Products are top-level components without interfaces.

Two languages describe the architecture. An *interface description language* (IDL) defines interfaces; a *component description language* (CDL) defines components and configurations. A *graphical representation* of CDL is used to create component diagrams. We find such diagrams to be of extremely high value in design discussions, for two reasons:

- They are *by definition* consistent with the implementation.
- They represent finer relations than component-component, while not going into the details of individual function usage.

Interfaces

As in COM, Koala defines interfaces as small groups of semantically related functions. We find it useful to categorize the interfaces between a component and its context as follows (see also Figure 5):

- *Provides interfaces* are the interfaces through which a component offers functionality to its context.
- *Explicit requires interfaces* are the documented and connectable interfaces that allow the user of the component to connect them to other components.
- *Implicit requires interfaces* are the non-connectable (and often undocumented) interfaces through which a component uses its environment.
- *Diversity interfaces* contain parameters through which the environment can fine-tune a component.

- *Resource interfaces* contain values through which a component can report to its environment how many resources it uses

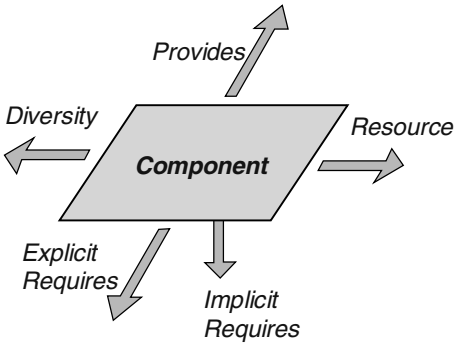


Fig. 5. Categories of interfaces

COM components have many implicit (and seldomly documented) requires interfaces. Koala encourages to route *all* functionality that a component needs through *explicit requires interfaces*, and to let a third party bind these interfaces at a higher aggregation level. The ideas are extracted from GenVoca [5] and Darwin [3]. Note that Szyperski [6] defines a component as *having explicit context dependencies only* and as *subject to composition by third parties*. Also note that in hardware, ICs only have explicit connection points.

Many people claim it's wiser to access standard 'computing platform' functionality through implicit requires interfaces, and to reserve the connectable requires interfaces for product variation. We don't agree for the following reasons:

- We want to *reduce* the context dependencies by *explicitly* monitoring it.
- We do not know all product variation in advance.
- Having all interfaces connectable allows us to *instrument* (trace, log, catch) function calls at all locations in the architecture (see [7]).

We are especially keen to avoid the *sticky toffee* (or *iceberg*) effect: trying to use a component from one application in another application usually reveals many (sticky) context dependencies. A lot of this context (the base of the iceberg) may have to be copied into the new application.

Diversity interfaces are used to parameterize components 'by name' (as opposed to e.g. the use of positional parameters as in Darwin). Diversity interfaces closely resemble property lists in Visual Basic. Resource interfaces are the opposite: instead of requiring values to be set by the environment, they inform the environment on the component's use of e.g. resources.

Diversity

There are two ways to implement two pieces of functionality that are alike but not the same. If the pieces are for 80% the same, we make it one component with a *diversity interface*, through which the component can be tuned into a product. If the pieces are

for 80% different, we implement them in *two different components* (some would call this component variants, but we just treat them as different components). If 50% of the required functionality is different, we split it up into one shared and two different components.

Product creation involves selecting the appropriate components, defining values for the diversity parameters, and binding them into products. This process of composition is executed recursively. Diversity parameters of inner components can be defined by outer components, or can be expressed in terms of the diversity parameters of outer components (or a combination of these two!). This process is also repeated recursively, resulting in what we sometimes call a *spread sheet of diversity*. Note that it would be too cumbersome to have to define *all* diversity parameters of *all* components at the top level!

If two compound components share a lot of internal structure but have some differences in binding (structural diversity), then a *switch* can be used to connect interfaces (the heptagon in Figure 4). The switch implements *conditional binding* (like an 'if' statement in Darwin), and the condition can be expressed in terms of diversity parameters of the outer component.

Finally, resource interfaces of components can be 'summed' and bound to diversity interfaces of other component supplying those resources. This allows for a certain degree of self-configuring systems. This configuration is mainly executed at compile time or initialization time.

Packages

In large development organizations, the technical notions of components and interfaces alone do not provide sufficient clarity for *who* is allowed to use *what*. From Java, we borrow the notion of *packages*, containers for component and interface definitions that allow certain definitions to be public and others to be private. Packages are developed by teams. Such teams can use private components and interfaces in the construction of their public components. Because they are private, changing them becomes a team issue, rather than a global issue. In the next sections we will see that we use packages to implement subsystems.

The Architectural Solution

We presented the technical solution in the form of a component model. But we need more to start large-scale development with this technology. We therefore defined a *global architecture* in terms of concepts, rules, and a global decomposition of the full functionality of the domain into subsystems that implement sub-domains. We cannot define a single variance-free or parametric architecture (see [8]) for the entire product population, since some of the products are just too different (a CD player and a TV have little in common). We therefore regard our subsystems as *reusable standard designs*, from which an arbitrary set of products can be created. This creation involves both binding and gluing!

The Global Architecture and Framework

We can only define relatively few rules at the global level. These include:

- The *component model* (Koala), supporting name spaces and configuration.
- The *code architecture*: the language C, and a predefined directory structure.
- The *documentation architecture*: data sheets and implementation notes.

Most other architectural concepts are defined in *regional* architectures (see below). Before we discuss that, we first define the notion of regions in terms of *subsystems* in a *reference architecture*.

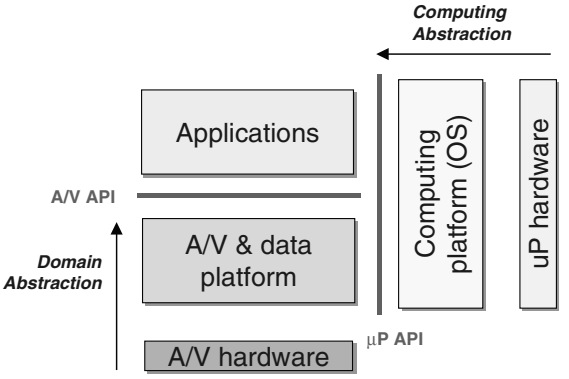


Fig. 6. The layered reference architecture

The Reference Architecture

Our reference architecture consists of a simple model that consists of three layers:

- software that abstract from the physical *computing platform*;
- software that abstract from the physical *audio/video processing hardware*;
- software that implement *services and applications* on top of this.

Note that this separation is not time-proof: we found that there is a tendency for functionality to shift over time from the A/V hardware domain via the applications domain into the computing abstraction domain. An example is audio streaming, which started using dedicated A/V hardware, then as computing capabilities grew became applications on general-purpose operating systems, and is now sometimes part of the operating system.

Figure 6 shows how we visualize the three layers to indicate that there are two independent axes of abstraction (right to left abstracts from computing hardware, bottom to top from domain hardware). The layers themselves are further split into subsystems as defined in the next section. Layers themselves are not subsystems or components; we just use them for a coarse classification of our subsystems into computing hardware dependent, A/V hardware dependent, and hardware independent.

Subsystems

In a single product, a subsystem is a large compound component that implements the functionality of a certain subdomain. A product consists of one to several dozens of subsystems; subsystems form the first level of decomposition.

Subsystems contain smaller components. The *definitions* of these components are reusable as well in our product family approach, but the *instances* are encapsulated by the subsystem (or more precisely, by the compound component). In other words, the instances of the smaller components can only be accessed through interfaces of the subsystem. Our component model ensures this.

In large-scale development, it is wise not only to encapsulate component *instances* but also certain component *definitions*! This allows a subsystem development team to use small components to build the subsystem, without being forced to freeze the definition of such small components. As implementation freedom, these components can be changed without notice. A language like Java offers such scoping in the form of *packages* with public and private entities, which we readily adopt (as described above). Not only component definitions but also interface definitions are handled this way.

In a product population, different products may need different implementations of certain subsystems. In the component model, this can be realized by a single compound component with diversity interfaces, or by multiple compound components (which can still have diversity interfaces!). The subsystem *package* can therefore contain more than one public (compound) component.

Finally, subsystem variation can sometimes also be expressed in terms of a single (large) compound component and a number of (smaller) glue components. A product architect instantiates the compound component and glues it to other subsystems using (a selection of) glue predefined components.

The terminology described above may be a little bit confusing. Just remember, from a development point of view, a subsystem is a *package of public and private component and interface definitions*. In a product, we often use the term subsystem to denote a single large compound component of this package.

Regional Architectures

Due to the large variety of products in our product population, many architectural concepts (and techniques) have a scope that is not global. To relieve subsystems for which the concepts are not relevant, we define these concepts in *regional* architectures.

Usually, a regional architecture involves one specific subsystem; we then speak of the subsystem architecture. Our infrastructure subsystem for instance, implementing the abstraction of the micro-controller as part of the computing platform, contains the concept of a *wake-up device*, which is a device that can wake-up the controller from a low-power standby state. This concept is mainly limited to the *infrastructure* subsystem, and is only described there.

Sometimes, a regional architecture involves more than one subsystem. An example is the software control of the audio and video hardware devices in the A/V platform. A number of concepts play a role here: power-up and down, exchange of signal measurement information between the software components, and persistency of

control values. But since the A/V platform is implemented by more than one subsystem, the concepts are defined in a regional architecture. Again, these concepts do not play a role in other subsystems, so they are not part of the global architecture.

The Process and Organization Solution

We discussed the technical and architectural solutions. Now it's time to describe the process and organization. As Figure 7 attempts to illustrate (by the way, this is also our logo), our approach embodies at any moment in time a *single common global architecture*, a set of *reusable subsystems*, and a set of *products* created with these subsystems. To evolve these assets over time, we define *projects* which we consider to be temporary activities, with a clear start and end point in time. They have the sole intention to 'newly or further develop' the architecture, one or more subsystems, or one or more products. The architecture, subsystems and products have a much longer life time than the projects.

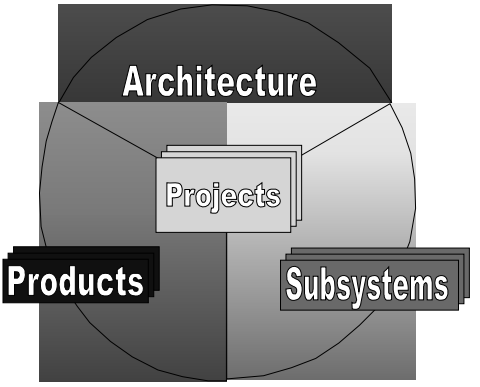


Fig. 7. Architecture, subsystems and products

We started our approach by assuming that a project develops either one subsystem or one product (or the global architecture). Soon, however, we found that the overhead associated with a project sometimes makes it necessary to combine the development of multiple subsystems into a single project. But we never let a project develop both a reusable subsystem and a product, to avoid conflicts of interest!

Documentation Architecture

A topic that most software architects consider to be of second order of importance is that of how to set-up the documentation of the software. For the development of a single product, indeed a classic approach of writing a client requirement specification, a software requirement specification, a global design, a detailed design, and then the implementation can be followed. In a product family (or population) approach, however, this does not provide the optimal structure to store information. We found it helpful to deviate in at least two ways from this 'common practice':

- We separate *interface* definitions from *component* descriptions.
- We do not write *requirement specifications*; we write *data sheets* (user manuals) instead.

Separating interface definitions from component descriptions seems cumbersome at first sight, but the added investment (of writing two documents) is earned back as soon as multiple components provide or require an interface of the same type. The idea of writing component data sheets is copied from hardware development. It describes the component from a user point of view, and not of a developer!

Next to these, we write architectural documents and component implementation notes in a more traditional way. Note that architectural documents are written at the global and regional level, and that implementation notes contain global and detailed design information, depending on the level of aggregation of the component in question. The statements above concern the *product* documentation; *project* information is documented in the usual way.

The Project Archive(s)

It is not uncommon for modern software projects to set-up three different archives:

- a source code archive
- a documentation archive
- a web site

We integrate these three archives into *one* single archive, which is automatically deployed on our company intra-net. This allows *all* developers in *all* sites access to *all* information. This 'open source' idea greatly enhances the communication between sites - we have no secrets for each other. Component orientation in fact induces this integration of archives: in our archive, each (lowest level) directory represents a component, with source code, documentation and web presentation!

Configuration Management

Configuration management (CM) systems are typically used for the following purposes:

- to manage a version history of files, components, et cetera,
- to manage *diversity* (platform and product variation),
- to manage *multi site development* (distributed configuration management),
- to control the *building* of products.

We limit the role of a CM system to the management of the version history alone.

- *Diversity* is explicitly managed in our component model (by creating different components, or by defining diversity interfaces), instead of 'hiding' it in a CM system;
- *Multi site development* is managed by true client-supplier relationships, as explained below.

- The *build process* is *controlled* by our software development environment, which allows us to choose state of the art solutions, instead of having to rely on the build facilities as delivered with the CM system.

Each subsystem is developed at a single site (but the site may develop more than one subsystem). The site does not share a CM system with other sites. Instead, the site releases its software in the form of ZIP files on the intra-net (after thorough testing, of course). Product development projects download the required versions of the required subsystems, and bind and glue their components together to form products.

One side effect worth mentioning is that we can build our applications on a notebook at home or in the plane (i.e. outside the scope of the CM system), which we find to be very convenient.

The Organizational Solution

Our software development is organized in terms of projects. As explained above, these projects either develop subsystems (design *for* reuse) or products (design *with* reuse). Each project is executed at a *single* development site. When allocating projects to sites, the capabilities of the sites are taken into account. These capabilities are explicitly managed, to improve the development performance on the long term.

Planning subsystem and product releases is done using a road map that shows the products of the coming few years, and the required functionality of subsystems. This roadmap is a living document; it is continuously updated to reflect the newest information. The person that is responsible for the subsystem projects is not also responsible for products, but instead reports directly to a senior manager in the organization.

Concluding Remarks

We have shown how we develop a large and diverse family of products. We use the term *population* to emphasize that the differences between products are so large that it is not feasible to define a large part of the architecture at a global level. Instead, our global architecture contains only the necessary elements for cooperation, and much of the architecture work is shifted to *regional* architectures. Other typical elements of our approach are:

- Our paradigm emphasizes *composition* over *decomposition*;
- We make *diversity* explicit in our architecture;
- We use a *client-supplier* relation between product and subsystem projects;
- We write *data sheets* (user manuals), instead of *requirement specifications*.

The approach is currently used to develop a product family of up-market televisions by more than one hundred people at five different sites. The development of a wide range of products (making it a true population) will start this year. But we are already observing that our approach indeed allows us to create a large variety of products, both in the way with which we can set-up test configurations for individual

components or subsystems, as in the way that we can create combination products in advanced development projects.

Many people have contributed to the ideas, their implementation, and my ability to present them in a (hopefully) clear way in this article. I would like to thank Pierre America, Hans van Antwerpen, Reinder Bril, Gerrit Muller, Henk Obbink and Jan-Gerben Wijnstra.

References

[1] Ivar Jacobson, Martin Griss, Patrick Jonsson, *Software Reuse – Architecture, Process and Organization for Business Success*, Addison Wesley, New York, 1997, ISBN 0-201-92476-5.

[2] Dale Rogerson, *Inside COM, Microsoft's Component Object Model*, Microsoft Press, ISBN 1-57231-349-8, 1997.

[3] Jeff Magee, Naranker Dulay, Susan Eisenbach, Jeff Kramer, *Specifying Distributed Software Architectures*, Proc. ESEC'95, Wilhelm Schafer, Pere Botella (Eds.) Springer LNCS 989 pp. 137-153 (1995)

[4] Rob van Ommering, *Koala, a Component Model for Consumer Electronics Product Software*, Proceedings of the Second International ESPRIT ARES Workshop, LNCS 1429, Springer Verlag, Berlin Heidelberg, 1998, p76-86.

[5] Don Batory, Sean O'Malley, *The Design and Implementation of Hierarchical Software Systems with Reusable Components*, ACM Transactions on Software Engineering and Methodology, 1 no. 4, pp. 355-398 (October 1992)

[6] Clemens Szyperski, *Component Software: Beyond Object-Oriented Programming*, Addison-Wesley, 1998, ISBN 0-201-17888-5.

[7] Robert Balzer, *An Architectural Infrastructure for Product Families*, Proceedings of the Second International ESPRIT ARES Workshop, LNCS 1429, Springer Verlag, Berlin Heidelberg, 1998, p158-160.

[8] Dewayne E. Perry, *Generic Architecture Descriptions for Product Lines*, Proceedings of the Second International ESPRIT ARES Workshop, LNCS 1429, Springer Verlag, Berlin Heidelberg, 1998, p51-56.

Requirements Modeling for Families of Complex Systems

Pierre America[1] and Jan van Wijgerden[2]

[1] Philips Research, Prof. Holstlaan 4, 5656 AA Eindhoven, The Netherlands
pierre.america@philips.com
[2] Philips Medical Systems, Veenpluis 4-6, 5684 PC Best, The Netherlands
jan.van.wijgerden@philips.com

Abstract. This paper introduces an approach to the specification of system families. The main ingredient of this approach is the definition of use cases hand in hand with a requirements object model. Instead of specifying individual systems, we specify a domain, i.e., a conceptual space of possible systems, in which individual systems can be defined by fixing a number of variation points. In that way we obtain a strong cohesion within the family and concise specifications of its members. We also describe a process suitable for this specification approach and indicate how the transition to the design phase can take place. Our approach was validated in one large project and several smaller ones.

1 Introduction

In many instances it is a good idea to conceive products to be introduced to the market in the context of a product family, whose members share several internal and external properties. Such a product family approach has many advantages, among others in the areas of marketing, development, manufacturing, and maintenance. During the development of such a family it is important to specify *what* is to be expected of these products in a sufficiently precise way so that all people involved (from marketing, development, etc.) agree on it without the possibility of significant misunderstandings. We call this requirements specification. (This is not to be confused with a specification of what is desired unilaterally, e.g., by the marketing department, or a specification of *how* this is realized.) Note that for a product family the requirements specification should make clear the properties of each individual member of the family while also making explicit what they have in common. Depending on the circumstances such a requirements specification could consist of a single page or of thousands of pages.

Furthermore, from a development point of view, we want to ensure a smooth transition from the requirements specification to the subsequent development phases (design, implementation, etc.). Among others it should be possible to use the documents developed for the requirements specification as a basis for the following phases, without, however, limiting the designers' freedom unnecessarily.

In this paper we describe a requirements specification method that is suitable for families of complex, software-intensive, electronic products. The following sections

F. van der Linden (Ed.): IW-SAPF-3, LNCS 1951, pp. 199-209, 2000.

describe different aspects of our method: the kind of development projects for which it is most suited, the documents that are produced, the way in which it supports product families, the specification process, and the transition to the design phase. Finally Section 0 briefly summarizes our experiences and Section 0 draws a conclusion.

2 Context

The approach described in this paper was applied, among others, in a large development project for a family of medical imaging equipment. Such a domain has the following characteristics:

- The market is relatively mature: Even the relatively modern systems with digital image processing have been around for more than ten years. Consequently, the expectations of the users are very high and to meet those expectations, the systems offer many possibilities to fine-tune them to the precise task at hand. This leads to a high degree of complexity.
- The radiation produced by the systems for the purpose of diagnosis is potentially dangerous to the health of patients and personnel. In addition, the high voltages necessary to produce X-rays, the strong magnetic fields necessary for MR (magnetic resonance), and the moving heavy machinery could constitute other hazards. Therefore high demands are placed on the safety and reliability of the equipment. This is one of the reasons why government institutions, such as the American FDA, apply strict rules not only to the equipment itself but even to the development process.
- The number of systems sold does not run in the millions, but in the thousands. Moreover there are many possible options to configure systems. As a result it rarely occurs that two systems have exactly the same configuration. Therefore it is impossible to test every possible configuration exhaustively.
- Developing such systems requires the cooperation of many people with vastly different expertise, ranging from VLSI designers to medical application specialists.

The project itself can be characterized as follows:

- It is large, involving over a hundred developers, many of whom were new to the domain.
- It is carried out jointly by several departments that previously developed their own product lines in relative isolation. As a result the different experts often used different concepts and terminology for comparable things.
- Although time to market is, of course, important, the complexity of the developed systems nevertheless leads to a project duration of several years. The resulting architecture should have a lifetime of well over ten years.

Because of the above factors, it is clear that an exhaustive and precise specification was necessary as a basis for further development.

3 Requirement Specification Documents

Specifying the required functionality of complex systems like these medical imaging systems in sufficient detail typically involves a large amount of documentation. Our method structures this documentation as follows:

- The **Commercial Requirements Specification (CRS)** describes the positioning of the system family members in the market. It only describes their features in very broad terms. By its nature, the CRS is primarily written by marketing experts, but it should be reviewed by developers to ensure its feasibility.
- The **Systems Requirements Specification (SRS)** sketches the features of the family members, typically in the form of bulleted lists and tables without much explanation of the terms used. This document and the ones below are typically written by developers and reviewed by other stakeholders.
- The **Functional Requirements Specification (FRS)** gives a complete and detailed description of the behavior of the systems in the family. Most of this is done in the form of use cases: pieces of English text that describe the interaction between the system and its users [5]. Despite its name, the FRS can also contain nonfunctional requirements, which can either occur in the use cases (e.g., acceptable response times for particular functions) or in a separate chapter. Since the total size of the FRS can be very large, it is often useful to divide the FRS into chapters, so that several authors can work on it in parallel. The method proposes to choose the chapters according to areas of functionality, for example, distinguishing different kinds of users or different phases in the user's workflow. For medical imaging systems, examples of chapter titles are Administration, Patient Positioning and Preparation, Acquisition, Reviewing, and Archiving. Often these functional areas coincide with areas of expertise for the several FRS authors involved. Note that this subdivision of the FRS does not automatically imply a similar subdivision in the design of the product family.
- A **requirements object model** expressed in UML [1]. This model explains the concepts that play a role in the interaction between the systems and their users and the relationships between these concepts. These concepts can range from concrete, tangible pieces of hardware (e.g., an X-ray detector), via electronic representations of real-world items (e.g., a patient folder) to abstract entities without a counterpart outside a computer (e.g., user preferences). This requirements object model provides a structured vocabulary for the FRS. In other words, all nontrivial words that occur in the use cases should occur in the requirements object model as names of classes, attributes, association roles, etc. Although the FRS is subdivided into chapters, there is a single, shared object model, which provides the common ground that ties the FRS chapters together and ensures that they talk about the same things.

The structure of a use case deserves some more explanation here, since it is slightly different from the literature [5]. For medical and similar professional electronic equipment, a typical use case involves only a single user. In most cases this user has a fairly simple interaction with the system (pushing a button, turning a knob, selecting a menu item, etc.), so the sequence of events is not complex enough to justify, for example, a UML sequence diagram. However, the effects, within and around the system, caused by this interaction need to be described very precisely to avoid

ambiguity. For example, when increasing the image contrast, it is important to specify precisely which set of images are affected by the change and whether the new contrast will still be effective when the images are viewed next time. This is where the connection with the model comes in as a conceptual framework to interpret these detailed descriptions. In principle, these descriptions could be expressed in a formal language, like OCL [9], but we prefer English because the specifications should be readable by different kinds of stakeholders, not only software developers. In fact, the goal is to write specifications that can be read as normal English texts by all people that know the meaning of the terms that are used.

Developing an object model during the requirement specification activity is a new element in our method compared with other approaches described in the literature [2] [3] [4]. We propose to develop the use cases hand in hand with the requirement object model (see also Section 0), combining to a certain degree the requirements specification and analysis activities of other methods. Although we do not exclude writing extensive use cases before any modeling is done, we do not consider the requirements specification finished until these use cases are expressed in the terminology defined by the object model.

Note that the construction of such a requirements object model is not completely comparable to what other methods call analysis. The difference is that often such an analysis is a first step in deciding *how* to implement the system's functionality, whereas our requirements model is intended to contain just enough information to express *what* the systems does. Although in practice, this distinction is not always completely clear, it is nevertheless much clearer than the distinction between analysis and design in many other methods.

The most important advantage of our approach is a significantly improved clarity and consistency of the FRS. While this could be achieved to a certain degree by including a flat glossary, the additional structure expressed in the object model by indicating the relationships of different items with each other increases this clarity and consistency. For example, if the model indicates that an X-ray system can have more than one X-ray detector, it becomes clear that a use case cannot just mention "the detector" but must indicate which specific detector is meant. Other advantages of the approach will be discussed below in relationship with the modeling and design processes.

4 Product Family Issues

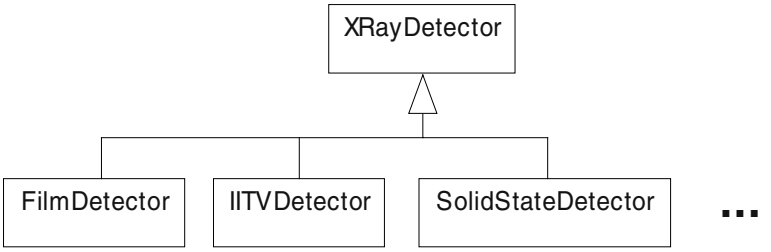
Since we are not specifying an individual system, but a whole family, the specifications should reflect that fact. This section describes how our method addresses this.

The most important principle is that, as much as possible, the specifications and the model should not describe one or more individual products, but a *domain*. Here we define a domain as a conceptual space of possible systems, which includes the current and future members of our product family. In other words, the specifications and the model should be as much as possible independent of the concrete choices for the individual products to be implemented as members of the family. In this way, the specifications and the model form a common basis for the whole product family, stressing the commonalties instead of the differences between the products. This

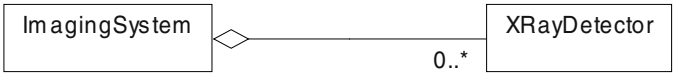
principle is important because it helps to eliminate inessential differences between the products that would otherwise arise naturally, especially when different people are responsible for different products in the family.

Nevertheless it is of course necessary to express the possible variation points between the systems in the domain. In the object model these can be represented in the following ways.

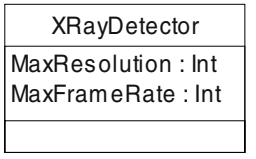
- Specialization: For a specific system, an object of a class in the model could be a member of a subclass. Whenever this is foreseen, such a subclass can already be included in the model, but this is not necessary. For example, an X-ray detector may be a film detector, a combination of image intensifier and TV camera, a flat solid-state detector, or any other kind of detector that may be invented in the future.



- Multiplicity: Aggregations or associations in the model can have ranges for their multiplicities, where the multiplicity for a specific system may have a single value within such a range. For example, the model may indicate that an X-ray system may contain one or more X-ray detectors, but of course any specific system will have a specific number of detectors.



- Attributes: A particular system may have a particular value for an attribute of an object in the model, taken from the type of the attribute in the model. For example the maximum resolution and the maximum number of images taken per second may vary among different detectors, even if they are of the same kind.



Note however that occurrences of the mechanisms above do not always correspond with system variation points: They could also apply to the data that is maintained by a single system. For example, a patient folder could contain an arbitrary number of study folders, and this number can even change over time within a single imaging system.

Variations in the use cases between the systems in the domain are such that each system supports its own subset of the use cases. Moreover, the behavior described in certain use cases may depend on certain parameters, which could differ between the systems. In this case, it is useful to include these parameters in the object model, typically as attributes. More generally, the use cases can depend on several kinds of items in the object model, e.g., the specific class of an object, the number and the identity of the objects involved in an association, or the value of an attribute. (This dependency can also occur if the model item is not related to a variation point between systems in the domain, e.g., a use case could depend on the number of study folders in a patient folder.)

In some cases, the straightforward application of the above mechanisms is not enough, but an in-depth, domain-specific analysis is required to capture the essential differences and commonalties between the systems in a domain. For example, while modeling medical X-ray systems, we applied this to the so-called *geometry*, the subsystem responsible for controlling the major moving parts in the system. The analysis showed that the essential concepts in the area of geometry were not motors, brakes, and C-arms, but positions and movements. All the systems in the domain shared the same structures for relative positions of X-ray sources, detectors, and patients. On the other hand, a movement could be characterized by two aspects:

- What is its clinical purpose? For example, rotating the X-ray beam around the patient, or moving a detector out of the way.
- In which way does the user control it? For example, the movement could be performed manually or by a motor.

On the basis of this characterization, the geometries of the systems in the domain could be distinguished by which clinical kinds of movements they support and by the control parameters of these movements (e.g., the maximum speed).

When all these measures have been taken, an individual system (possibly a product in the family) can be specified by determining the set of supported use cases and fixing the variation points in the object model (i.e., specifying the subclass in a generalization, the multiplicities of an association, or the value of an attribute).

During the specification process it may be helpful to consider a small number of specific systems, which may be actual envisaged members of the product family or fictitious ones. By doing this, we can ensure that the abstractions in the FRS and the model actually fit the intended concrete and specific instances. These specific systems can also be included in the model, but in that case they must be clearly marked as examples.

5 The Requirements Specification Process

We propose to involve as many stakeholders as possible in the requirements specification process, since this will increase subsequent acceptance of the specifications and the underlying model. Compared to the CRS and the SRS, the involvement of developers in the FRS and object model will be typically be higher, whereas the involvement of marketing people will typically be lower. For the FRS authoring and object modeling we have good experiences with the following team structure:

- An FRS authoring team for each chapter of the FRS. Such a team will consist of the primary author of the FRS plus a number of supporters, which can be experts in the field or representatives of stakeholders. The responsibilities of this team are to write the FRS chapter and to contribute to the requirements object model.
- A single object model control team (OMCT). This team consists of a small number of people (three or four), including at least one person with experience in modeling and possibly a designated scribe. This team does not need much domain knowledge to begin with, since the FRS authoring teams will contribute that. The responsibility of the OMCT is to maintain the object model and to ensure its internal consistency.

Modeling can begin when the people in the FRS team have a good, but informal understanding of the required functionality. Often they have acquired this familiarity by being involved in previous development projects for similar systems. When necessary this understanding can be enhanced by writing and discussing informal use cases (along the lines of [5]).

In order to obtain a single, consistent object model that provides a basis for all the FRS chapters, the model is constructed by overlapping modeling teams, where each modeling team consists of one FRS authoring team plus the OMCT, supplemented by additional experts whenever needed. Each such group has an initial modeling session, typically lasting one or more weeks, in which the members of the group share their domain knowledge and together construct a piece of the object model (which should be consistent with earlier pieces, of course). Then FRS authoring can begin on the basis of this initial object model and in the following period, return sessions with the OMCT are held, e.g., a half or a full day per week, in order to fine-tune this model. The typical changes to the model during this fine-tuning fall in the following categories:

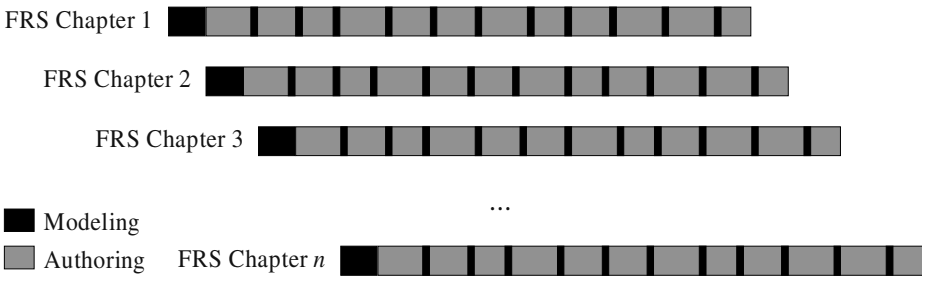
- While writing a use case, it turns out to be difficult with the concepts in the existing model, so the model is changed to solve this problem.
- A more elegant way has been found to express a certain point in the model.
- Another modeling group has proposed a change in the model and this must be integrated with the use cases written by the current modeling group.

Modeling outside these modeling teams should be discouraged, since in the subsequent modeling sessions this tends to lead to a discussion about the merits of different solutions, which is more difficult than a discussion about a problem and how to solve it.

Here each horizontal bar represents the activities of a single FRS authoring team. The black parts represent modeling sessions, which cannot overlap because the OMCT must take part in each of them. The authoring work, indicated by gray parts can take place in parallel with the other teams.

In such a time schedule, the OMCT is a shared resource, which could become a bottleneck. In practice this danger can be mitigated by clever planning of the sessions, since typically the people in the FRS teams do not all become available at the same time. However, a strong continuity in the staffing of the OMCT is important, since its members must be familiar with the complete object model and have a reasonable overview of the contents of the FRS chapters.

The following picture illustrates the overall time schedule.



The advantage of the above process is that a lot of domain knowledge is exchanged and shared early in the development process, even though only a part of this knowledge can be consolidated in the object model or the FRS. (In any case, this consolidated part is much larger than when working with a flat glossary.) Modeling in overlapping groups has the following advantages:

- Many people can be involved in the process, which increases the quality, completeness, and acceptance of its outcome.
- A lot of work (writing the FRS chapters) can be done in parallel.
- The modeling itself takes place in manageable groups.
- The overlapping part (the OMCT) can ensure consistency, not only in the contents of the model but also in modeling style.

6 Towards a Design

The requirements specification artifacts described above serve as a solid starting point for the design workflow. One output of the design workflow is a design object model. The first step towards such a design model is a purely mental one, in which not the model itself, but its interpretation is changed. For example, a UML class in the model represents a (real-world or imaginary) concept when considered in the requirements model, but in the design model the same UML class represents a programming language class. Similarly, an attribute of such a class represents a property of the corresponding concept in the requirements model, but in the design model it stands for a set of access methods and possibly a data member. Of course, this step towards an initial design model does not involve any new artifacts, because the description of the requirements model can simply be reused, but probably because of this, the mental step is a difficult one for many developers and it takes some time to get used to. More on this issue can be found in [6], although the final approach described there is different from ours.

The second step towards a design model consist of iteratively doing one of the following:

- Extension: Adding packages, classes, attributes, operations, etc., as necessary.

- Subdivision: Grouping classes into packages. (Attributes and operations are already assigned to classes when they are introduced.)
- Assigning responsibilities: Every package, class, and operation should have a clear and concisely described responsibility.

These steps are carried out under guidance of an architecture, which especially addresses the non-functional requirements. Of course, a lot more can be said about how to carry out these steps in order to arrive at a design that is optimized for a product family. That is outside the scope of this paper, so for more information we refer the reader to [7], [8], and [10]. However, we should mention here that feedback towards the requirements specification process should make sure that in the unlikely case that the requirements cannot be implemented within the constraints of the development project, the FRS and possibly the object model are changed by the FRS authors and the OMCT.

In any case, the result of these steps is a design object model that is a pure extension of the requirements object model. The most important advantage of this is the enhanced traceability: The relationship between the items in the requirements model and the ones in the design model is clear.

At first sight, insisting that the requirements model is a pure subset of the design model seems a very strong constraint: While the requirements model is constructed, little or no thought is given to the design, and still the requirements model shapes the design to a significant degree. Nevertheless, the following two principles make sure that the designers have enough freedom to come up with an effective and efficient design:

- The requirements model contains only those items that are necessary to explain *what* happens in the system. The items related to *how* things happen (e.g., control and interface classes) do not belong to the requirements model and can therefore be added in complete freedom by the designer.
- Responsibilities are not assigned to items in the requirements workflow, but this is left to the design. This again provides the designer with the freedom to assign the responsibilities in a suitable way. For example, when the requirements model says that a class has an attribute, this does not mean that the class also has the responsibility to store the value of that attribute. Instead, the value can be stored somewhere else or computed whenever needed.

7 Experience

As already mentioned in the introduction, our method was applied and refined in a large development project for a family of medical imaging systems. The requirements object model for that project became quite large, containing about 100 diagrams, 700 classes, 1000 relationships and 1500 attributes. The general feeling in the project crew is that this way of requirements modeling and specification laid 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. The design effort was firmly based on the requirements model, which did end up as a submodel of the design model. It is

true that many designers, especially the ones that were not involved in requirements modeling, had a natural tendency to deviate from the original model, an open technical discussion always led to good designs that did extend the requirements model.

Several smaller feasibility studies have confirmed that this approach is suitable over a wide range of professional electronic systems.

8 Conclusion

In this paper we have described a method for specifying requirements for a family of products. This method involves the construction of a requirements object model along with the use cases that describe the functionality of the products. The result of this method is a fairly precise specification for the whole domain, which can be complemented with quite concise specifications for the individual products. We have applied this method successfully to one large project and several small ones.

The most important things we have learned from this are the following:

- The construction of a requirements object model early in the development process is very useful, especially in a large project, because it provides an explicit, shared map of the concepts playing a role in the domain and that map can be used through the whole development effort.
- Developing the use cases and the requirements model hand in hand leads to more precise formulations of the use cases and provides a validation of the completeness of the model, while still leaving maximal freedom for the subsequent design phase.
- Our way of constructing the model in overlapping groups allows many people to participate in the modeling, while at the same time keeping the model consistent and facilitating a considerable amount of work to be done in parallel.
- The most important aspect is not the individual technique used for specification, modeling, designing, etc., but the way in which all the techniques fit together in the overall development process.

Although we have tried this approach only on a variety of complex software-intensive electronic systems, we envisage that it may well be applicable to a much larger range of software systems.

Acknowledgements
This research has been partially funded by ESAPS, project 99005 in ITEA, the Eureka Σ! 2023 Programme.

Many people at Philips Medical Systems, Philips Research, and other parts of Philips have contributed to the conception and refinement of our method by their cooperation and support. They are too numerous to mention here but that does not diminish our gratitude. Herman ter Horst and Jan Gerben Wijnstra gave valuable comments on earlier versions of this paper.

9 References

[1] Grady Booch, Ivar Jacobson, and James Rumbaugh: The Unified Modeling Language User Guide. Addison-Wesley 1998.
[2] Bruce Powel Douglass: Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks and Patterns. Addison-Wesley 1999.
[3] Desmond F. D'Souza and Alan C. Wills: Objects, Components, and Frameworks with UML: The Catalysis Approach. Addison-Wesley 1998.
[4] Ivar Jacobson, Grady Booch, and James Rumbaugh: The Unified Software Development Process. Addison-Wesley 1998.
[5] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Overgaard: Object-Oriented Software Engineering: A Use Case Driven Approach. ACM Press/Addison-Wesley 1992.
[6] Hermann Kaindl: Difficulties in the Transition from OO Analysis to Design. IEEE Software, pages 94–102, September/October 1999.
[7] Henk Obbink, Rob van Ommering, Jan Gerben Wijnstra, and Pierre America: Component oriented platform architecting for software intensive product families. Symposium on Software Architectures and Component Technology, Enschede, The Netherlands, January 20-21, 2000. Kluwer Academic Publishers.
[8] Ben J. Pronk: Medical Product Line Architectures – 12 years of experience. Proceedings of the First IFIP Working Conference on Software Architecture, 1999.
[9] Jos B. Warmer and Anneke G. Kleppe: The Object Constraint Language: Precise Modeling With UML. Addison-Wesley 1999.
[10] Jan Gerben Wijnstra: Component Frameworks in a Medical Imaging Product Family. Third International Workshop on Software Architectures for Product Families, Las Palmas de Gran Canaria, Spain, March 15-17, 2000 (this volume).

Creating Product Line Architectures[1]

Joachim Bayer, Oliver Flege, and Cristina Gacek

Fraunhofer Institute for Experimental Software Engineering (IESE)
Sauerwiesen 6, D-67661 Kaiserslautern, Germany
{bayer, flege, gacek}@iese.fhg.de

Abstract. The creation and validation of product line software architectures are inherently more complex than those of software architectures for single systems. This paper compares a process for creating and evaluating a traditional, one-of-a-kind software architecture with one for a reference software architecture. The comparison is done in the context of PuLSE-DSSA, a customizable process that integrates both product line architecture creation and evaluation.

1 Introduction

Product line engineering is an approach to improve development efficiency for families of systems by facilitating large-scale reuse. It typically focuses on building a reuse infrastructure that can then be used to derive the product line members. A core asset of such a reuse infrastructure is a product line architecture (also known as reference architecture or domain-specific software architecture). Software architectures are a set of components, connectors, constraints imposed on the components, connectors, and their composition, and a supporting rationale. They are presentable in various ways — different views supporting different needs [4]. A product line architecture is a software architecture for supporting a complete product family, it reflects common parts as well as variabilities among the various products. Product line architectures define the essential parts of the reuse infrastructure and thus ensure that shared and instance-specific components fit together for all product line members.

In this position paper, we illustrate the specific aspects of architecting for product lines in the context of PuLSE-DSSA, the reference architecture development component of the PuLSE™ (Product Line Software Engineering)[2] method [2]. PuLSE is a method for enabling the conception and deployment of software product lines within a large variety of enterprise contexts. To achieve this, the different PuLSE components are customizable to different situations and contexts.

---

[1]This work has been partially funded by the ESAPS project (Eureka Σ! 2023 Programme, ITEA project 99005).

[2]PuLSE is a registered trademark of the Fraunhofer IESE

F. van der Linden (Ed.): IW-SAPF-3, LNCS 1951, pp. 210-216, 2000.

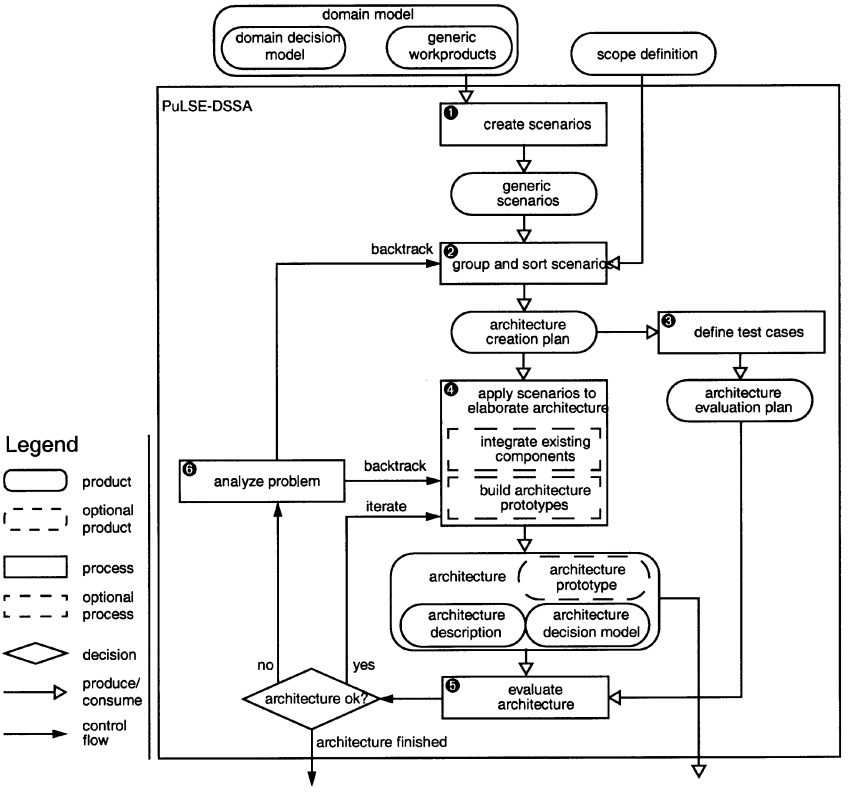


Figure 1 PuLSE-DSSA Process Overview

2 PuLSE-DSSA

PuLSE-DSSA is based on the generic architecture development process shown in Figure 1. This process is generic because it abstracts from the differences between one-of-a-kind and product line architecture development, and also from the customizations necessary to use this process in different application contexts. The only product line specific aspects shown in Figure 1 are the input and output products of PuLSE-DSSA. The input is a scope definition and a domain model, where the former defines the business case for the development of the product line and the latter describes commonalities and variations of applications within the product line. Output of PuLSE-DSSA is a product line architecture as defined in the introduction.

In the following subsections, we present for each process step a generic description and then point out the product line specific aspects, including customizations, that have to be taken into account when developing a reference software architecture.

2.1 Create Scenarios

The first step in architecture creation is to determine the most important requirements. These are captured in scenarios that describe critical use-cases (i.e., how the system is used to perform a specific task) as well as system level quality objectives (i.e., non-functional requirements) and constraints. This step is necessary for a generic process in order to decouple it from the various kinds of inputs (i.e., requirements specifications) that are available in different contexts.

Product Line Aspects

In the PuLSE product line development process, the input for this step would be a product line model consisting of generic workproducts (i.e., products describing requirements in terms of commonalities *and* variabilities) and a decision model.[3]

Conventional scenarios are described on an instance level, which makes it difficult to convey the variability information needed for product line requirements and to extract the information that applies to a particular instance. Therefore, it is beneficial to use generic scenarios that represent commonalities and variabilities like the product line model's generic workproducts, and are also instantiable via the domain decision model.

Managing traceability information is important to achieve effective maintenance and consistent change management. This is even more valid in a product line context, because a product line infrastructure is a strategic investment and will almost certainly be maintained for a long time. As a first step towards full traceability, the scenarios have to be linked to elements of the product line model.

2.2 Group and Sort Scenarios

This step yields the architecture creation plan that defines the iterations in which the architecture development is performed. The first iteration deals with the most important group of scenarios, the second one with the second most important group and so forth. The order in which scenarios are addressed is highly significant, because each iteration's design decisions impose constraints on the architecture that delimit its further evolution. Unfortunately, grouping and ordering of scenarios is also a highly subjective task, as it relies on a combination of domain knowledge (ability to judge the importance of a scenario) and system architecting experience (ability to anticipate how a scenario could affect the architecture).

---

[3] A *decision model* captures variability in a product line in terms of open decisions and possible resolutions. In a *decision model instance*, all decisions are resolved. As variabilities in generic workproducts refer to these decisions, a decision model instance defines a specific instance of each generic workproduct and thus specifies a particular product line member.

Product Line Aspects
The judgement of a scenario's importance cannot be based on its expected impact on the architecture alone. A complementary factor is the overall importance a scenario has for the product line. Therefore, the information contained in the scenarios is insufficient and has to be augmented with the scope definition. Ideally, the scope definition is based on a process that seeks to estimate the economic value each distinct feature would have for the product line (e.g., PuLSE-Eco [3]). Assuming that economic value and importance correlate, some of the subjectivity of this process step can thus be reduced.

2.3 Define Test Cases

For each group of scenarios, test cases are defined that will be used to evaluate the architecture at the end of each iteration. It should be possible to conduct the tests automatically in order to facilitate regression testing, which is particularly important for an iterative development process. Specifying test cases *before* the actual development begins has a number of benefits, including a better understanding of the requirements and avoidance of creating tests that, due to a fixed perspective, merely support what has been developed.

It is important to note that evaluation needs have an impact on the choice of how the architecture should represented, because different notations allow for different kinds of (automatic) evaluations.

Product Line Aspects
In a product line context, instance- and family-specific test cases have to be distinguished. The former do not differ from those used in one-of-a-kind architecture development and are therefore less of a problem. The latter focus on specific kinds of system level quality objectives such as maintainability, understandability, and reusability. These qualities play a critical role for the appropriateness of a reference architecture. However, it is extremely difficult to assess the degree to which theses qualities are achieved by a given architecture. This problem is hardly addressed by any architecture evaluation method (e.g., SAAM[4] [5], ATAM[5] [6]) nor by any architecture description language [7]. Neither have – to our knowledge – any guidelines been published on how testing should be performed to ensure that a product line infrastructure will be adequate.

[4] SAAM assesses the impact of anticipated changes (new requirements) to predict modifiability. The idea behind product lines is to consider anticipated changes already during architecting. Therefore, we should expect that a SAAM scenario applied to a product line architecture is satisfied by a particular instance of that architecture without needing any noteworthy changes

[5] A TAM provides a framework for evaluating architectures, but does not provide the methods that are required to use the framework (e.g., once you have a method to describe and assess the understandability of an architecture, you could use ATAM to investigate the trade-offs between understandability and other quality attributes).

2.4 Apply Scenarios

The group of scenarios associated with the current iteration is used to create the initial architecture or to refine/extend an already existing, partial architecture. This step also includes the possible integration of existing components (legacy or COTS) as well as prototyping. The result is a (partial) architecture description and possibly a prototype. During the application of scenarios, it is important to capture design decisions and link them as well as architectural elements to scenarios to ensure traceability.

Product Line Aspects
During architecture development some variabilities might become apparent that are not driven by the problem domain, but rather by the solution (e.g., two components addressing the same problem, yet implementing differing algorithms). In this case, the domain decision model is complemented by an architecture decision model. All open decisions in both of these models have to be resolved for instantiation.3w

The following issues address possible customizations of this step. First, an appropriate representation for the product line architecture has to be chosen, which is a major problem, because mainstream notations (e.g., UML) and tools do not support the description and instantiation of generic architectures. Furthermore, different parts of the architecture might require different representations as described by Perry [8].

Hand in hand with choosing a representation goes choosing (creating) an instantiation mechanism and process. This may even encompass the construction of the necessary instantiation infrastructure (tools, etc.). By defining how instantiation is performed, it is also determined how variability is actually *implemented*.

Finally, the extent to which prototyping is performed must be defined. As illustrated earlier, a theoretical groundwork for evaluating reference architectures based on models (descriptions) is almost not existing, which suggests that prototyping is even more important than for one-of-a-kind architecture development.

2.5 Evaluate Architecture

In this step, the architecture resulting from the previous step is evaluated according to the architecture evaluation plan. If the evaluation is successful (i.e., all tests are passed), the architecture development continues with the next iteration or is finished once the last group of scenarios has been applied. If, however, at least some test failed, the process continues with step 2.6 “Analyze Problem”.

Product Line Aspects
Evaluation has to address instance-specific as well as family-specific aspects and relies on a defined instantiation mechanism. First, the *ease* with which instances can be created allows to draw conclusions on critical family-specific characteristics (i.e., usability, maintainability, etc.). Second, the range of possible instantiations is used to ensure that the intended products are indeed covered by the reference architecture. Third, instantiation yields variability-free architectures that can be evaluated in the same way as one-of-a-kind ones. It is obvious that it is difficult to both get and interpret these results without relying on a prototype and on an automatic instantiation mechanism.

A lot of instance-specific tests tend to be applicable for several or even all instances. It is therefore sensible to use this process step as a starting point for the construction of an infrastructure for testing and debugging architectures of product family members as proposed by Balzer [1].

2.6 Analyze Problem

At least one of the tests for evaluating the current architecture failed. In this step, the underlying problem is examined in order to determine how the architecture development process can be continued. The examination focuses on whether the current group of scenarios could be applied successfully to the architecture that resulted from the previous iterations. If this is deemed to be the case, only the current iteration needs to be reiterated. Otherwise, some design decisions from an earlier iteration are presumed to impose constraints that are too stringent for the current set of scenarios. Therefore, extended backtracking is needed, which may include reformulating, regrouping, and reordering of some scenarios and then reentering the process in the appropriate iteration.[6]

Product Line Aspects
The task of finding out whether a given, partial reference architecture is compatible with a new set of requirements is significantly more difficult than to judge that for a one-of-a-kind architecture. Ensuring that an addition does not break any of the possible instances of the architecture is a highly complex task, especially when it involves dealing with entities that could not be encapsulated properly. It is noteworthy that this problem is not only relevant for architecture creation but also for maintenance. We are not aware of any published guidelines on how this problem should be addressed.

3 Summary and Future Work

In this paper we have presented the PuLSE-DSSA method, while discussing its customizability and how it is to be applied for the creation and evaluation of product line architectures. This discussion was used to highlight how methods supporting product line architectures must deal with complexities non-existent in those for one-of-a-kind software architectures.

PuLSE-DSSA presents the framework for most of our current research in software architectures. Some of the issues we are currently working on include means of representing product line architectures with their commonalities and variabilities; the derivation of instance architectures from a given product line architecture; supporting traceability between architectural elements and various other assets; and the evaluation of product line architectures.

---

[6] In some situations backtracking to the requirements definition phase (e.g., domain modeling) may be necessary. This involves change management processes that are outside the scope of this paper but are addressed within PuLSE.

References

1. R. Balzer, "An Architectural Infrastructure for Product Families," in *Proceedings of the Second International Workshop on Development and Evolution of Software Architectures for Product Families*, Lecture Notes in Computer Science 1429, pp. 158-160, Springer, 1998
2. J. Bayer et al., "PuLSE: A Methodology to Develop Software Product Lines," in *Symposium on Software Reusability'99 (SSR 99)*, pp. 122-131, May 1999
3. J.-M. DeBaud and K. Schmid, "A systematic approach to derive the scope of software product lines," in *Proceedings of the 21st International Conference on Software Engineering (ICSE 99)*, pp. 34-43, 1999
4. C. Gacek, A. Abd-Allah, B. Clark, and B. Boehm, "On the Definition of Software Architecture," in *Proceedings of the First International Workshop on Architectures for Software Systems*, D. Garlan (ed.), Seattle, WA, pp. 85-95, 24-25 April 1995
5. R. Kazman, G. Abowd, L. Bass, P. Clements, "Scenario-Based Analysis of Software Architecture," *IEEE Software*, vol. 13, no. 6, pp. 47-55, November 1996
6. R. Kazman, M. Barbacci, M. Klein, S.J. Carriere, and S.G. Woods. "Experience with Performing ArchitectureTradeoff Analysis," in *Proceedings of the 21st International Conference on Software Engineering (ICSE 99)*, pp. 54-63, 1999
7. N. Medvidovic, "A Classification and Comparison Framework for Software Architecture Description Languages", *Technical Report UCI-ICS-97-02*, University of California, Irvine, CA, Feb. 1997
8. D. Perry, "Generic Architecture Descriptions for Product Lines," in *Proceedings of the Second International Workshop on Development and Evolution of Software Architectures for Product Families*, Lecture Notes in Computer Science 1429, pp. 51-56, Springer, 1998

Extending Commonality Analysis for Embedded Control System Families

Alan Stephenson[1], Darren Buttle[1], and John McDermid[1]

Department of Computer Science, University of York
Heslington, York YO10 5DD, UK
{alan.stephenson,darren.buttle,john.mcdermid}@cs.york.ac.uk

Abstract. Families of systems are prevalent in software engineering, and many techniques are available for the analysis and implementation of such families. Embedded systems families provide additional constraints on these techniques, and safety-critical systems families constrain them even further. An existing family analysis technique is combined with a layering scheme to structure the selection of variations and the introduction of detail within those constraints. Application of this new technique produces software which is specified and configured during development, within a fixed execution architecture which reflects the common aspects of systems within that family.

1 Introduction

Scope, Commonality, Variability (SCV) analysis has been proposed as an effective means of analysing a family of similar systems[2]. The analysis produces a description of common system aspects, and of the variations that make each of the family members unique. By undertaking such an analysis, an organisation can better prepare itself for the evolution of the family product line as a whole.

In the embedded systems domain, families of systems have a number of properties which make a commonality analysis less straightforward. Firstly, an embedded system operates within a larger engineering environment, and is typically developed concurrently with that environment. This situation involves two families – the family of embedded systems, and the family of surrounding (embedding) systems. Members of these families are integrated so fully with one another, that the families are interdependent. A member of one of the families cannot operate without a corresponding member of the other.

Additionally, many embedded systems carry real-time or safety-critical constraints. These concerns have many implications for the family development process. The process must deal with the timing and safety analysis activities that are used to justify the correct operation of the embedded system, and so any embedded software system must be amenable to such analysis – often to the satisfaction of the appropriate safety certification authority. The techniques available for the implementation of a safety-critical embedded system are limited by this type of analysis, and do not extend to the dynamic, object-oriented implementation strategies currently used in product lines.

F. van der Linden (Ed.): IW-SAPF-3, LNCS 1951, pp. 217–224, 2000.

One such embedded systems family is the family of engine control systems developed by Rolls-Royce plc. The development of an engine controller to flight standard is a costly process, and any reduction of these costs must be achieved without compromising existing engine controller safety. Each controller is developed for a particular variant of a particular engine series. For some years, Rolls-Royce plc. have supported a number of University Technology Centres (UTCs) to work on key aspects of engine design and technology. A UTC in Systems and Software Engineering was established in York in 1993. The work described here is a central part of a programme intended to improve the engine controller software engineering process by introducing a product family approach.

2 The Product Line

To serve as a focal point for this process, Lucent's SCV analysis was considered. It was evaluated with respect to the "sub-product line" of aero-engine thrust reverser systems, and this analysis was successful in a number of ways:

- The amount of commonality between these systems was high, estimated at around 80%.
- The identification of commonalities at a high level of abstraction would provide a good way of partitioning the system into components.
- Variabilities would sometimes prompt ideas about the way in which the family could evolve in the near future.

Some difficulties were uncovered, though:

- The influence of externally-applied requirements on the variabilities was not easily seen in the SCV scheme.
- The complexity of safety-critical systems such as the thrust reverser made cross-referencing between commonalities and variabilities difficult to use.
- Safety analysis is difficult to apply without a clear separation of concerns and responsibilities.
- The separation of information according to domain issues did not become clear until after a number of analysis iterations.

To address some of these concerns, ideas on layering, from the existing research on the requirements engineering process within the UTC, and on representations of variations among requirements, from Napier University[4] were integrated into a new commonality-based embedded systems family development process.

2.1 Layering

Previous research at the University of York has advocated a layered software engineering process[6]. Requirements presented at the topmost layer are mapped into a design, which is analysed to produce a new set of requirements for the

next layer. This process is used to provide traceability between requirements and design decisions, both for safety analysis and for product evolution.

The layering introduced into the commonality-based development process operates in a similar fashion, but separates out externally-applied requirements across all of the layers. This allows each layer to represent a uniform level of abstraction, capturing the progress of development from a set of requirements to a design which meets those requirements.

The designs within each layer are not necessarily software systems, however. Around the embedded system, there is a larger concurrently-engineered environment with which the embedded system must operate. That environment is also modelled within each layer, and the decisions made during its design affect the way in which the software design can discharge its requirements. In principle, the same is true in the other direction, although this flow of information is rarely seen in practice.

Within each layer, therefore, there is a traceability between the externally-applied requirements and the designs of the concurrently-engineered systems, of which the software is but one. This design information is carried through into the following layer, until a final design is reached – physical components in the surrounding system, or software implementations in the embedded system.

2.2 Variability Notation

A structural notation has been derived directly from work carried out by Mannion and Keepence at Napier University[4]. Our interpretation of variation dependencies is represented with variation operators, given in Table 1.

Table 1. Variation operators

Category	Operator	Description
Commonality	CO(Item)	The item is selected in each instance of the family.
Variability	VO(Item)	The item may or may not be selected for a particular instance of the family.
Parameter	PO(Item)	The item quantifies an aspect of the family.
Choice	VCL(Items)	The items are mutually exclusive. Exactly one of them is selected for an instance of the family.
Selection	VSL(Items)	The items may be combined. At least one of them is selected for an instance of the family.

In addition to these variation operators, tracing clauses show the requirements discharged by each design element, and derivation clauses show how items in one layer relate to those in previous layers. Dependency clauses are added to indicate when an item should be considered for selection. These clauses can be used to group a number of commonalities under a single variability, so that those

commonalities are only common to instances where the variability is selected. Equally, a number of commonalities and variabilities can be derived from a single commonality at the previous level of abstraction. These two situations are outlined in Figure 1. The flexibility of these clauses is such that one variability might only be considered for selection when another had been considered but not selected, for example.

Fig. 1. (a) Elements dependent on a variability. (b) Elements derived from a commonality.

3 Building a New Process

By using both of these techniques together, a number of benefits can be seen, some of which directly address the concerns outlined above:

- Things which are common at one level of abstraction can vary when more detail is added. The layering helps to remove some of the complexity in maintaining cross-referencing information.
- Domain-specific aspects surface when variabilities at the lower levels become grouped under commonalities at higher levels. This grouping occurs as soon as selection dependencies are considered, and can help the analyst to think in terms of domain-specific abstractions.
- Requirements imposed on the project externally are separated from information about concurrent designs, which shows how the requirements are being met with more precision than a more conventional software development scheme. This is beneficial for the safety analysis process, but also allows the analyst to track changes in responsibility between the different concurrently-engineered systems and the external customer.

4 Linking to Software Architecture

Each layer of the analysis will contain a number of concurrent designs, in different disciplines. One of those disciplines is software engineering, and abstractions of software designs are typically conveyed using software architecture. The software architecture represents the allocation of design functionality to discrete design elements, and adds information about the relevant nonfunctional properties. For

embedded systems families, timing, synchronisation and fault accommodation are the major nonfunctional concerns.

A prototype domain-specific architectural style has been introduced in order to deal with the representation of some of these nonfunctional properties. It takes some inspiration from the graphical notation of MASCOT[3,5], coupled with the object classification scheme of HRT-HOOD[1]. Synchronisation of information flows is represented using a modified dataflow scheme, allowing for various patterns of synchronous and asynchronous interaction.

In most architectural styles, the transformation that adds detail to the specification is a straightforward refinement. In this scheme, however, conventional refinements are made more difficult by the interactions between the embedded system and its embedding environment, and the need to control that environment safely. It is necessary to appeal to safety analysis, timing analysis, and the design of the embedding system whenever design decisions are being made. For example, a safety analysis of the specification, even at a low level of detail, might indicate that a watchdog timer is an appropriate safety measure. An interface to the watchdog must therefore be added to the software, and additional timing constraints might also be imposed.

5 Family-Based Architecture

The software architecture of a number of instances of a family of systems can itself be analysed for its commonalities and variabilities, and stored as the software design within the appropriate layers. However, to make better use of the family-wide information base, a different approach is proposed.

Previously, it was noted that commonality analysis will usually identify domain concepts which represent the common elements of the domain at its highest level of abstraction. This information can be used to suggest a decomposition of software designs into architectural elements which also represent those common elements. This structuring approach produces an architecture for the entire family, rather than for an individual member of that family. Operating across the family in this way has a number of distinct advantages:

- The same software architecture is present for each system within the domain. Each one uses the same notations to represent the same organisation of components, making it easy to understand the family.
- The design elements in each layer are organised according to commonality and variability, as well as by relationship to previous layers. It is easy to see how the architecture varies to support the variation within the domain.
- When a number of variabilities are derived from a commonality in a previous layer, they are represented by a single architectural component. The effects of these variations are localised to that architectural component.
- The architectural style specifies interfaces between components at all levels. Changes within one component which do not violate that interface should not propagate beyond that interface. This allows the component boundary to act as a change containment boundary.

– The analysis itself can be used as a partial justification for the structure of the software, providing rationale based on the frequency of changes[1].

With design information captured in this way, individual software implementations can be generated by selecting the appropriate architectural variants through the successive layers. Deciding on, and creating, the appropriate variants becomes the major engineering task in the development and evolution of the system. While the creation of the software itself is firmly in the hands of the software engineer, the selection of variants can be traced from selections made among the family of embedding systems, and the family of requirements for those systems.

Once the software has been selected and built for a particular member of the embedding systems family, its structure is fixed – nothing is allocated or deallocated during execution. To reconfigure the software, it must be reselected from within the family and rebuilt. This provides a flexible family-based approach, but also enables the use of static analysis techniques which a more flexible implementation would prohibit.

6 Further Research

We are currently investigating the use of this process through an extensive demonstrator programme, in conjunction with Rolls-Royce plc. This programme aims to show that a product family development strategy will significantly reduce development costs within that family, by developing representative software systems for a subset of the engine controller family.

The first part of the demonstrator programme includes the commonality analysis, and provides the product family's domain-specific architecture. In the second part of the programme, one family member is developed, and then modified to produce another family member. It is expected that the links between commonalities and variabilities will provide most of the software elements, and will, for example, allow the rapid identification of requirements for which no software elements yet exist.

To fully support this programme, however, there are still a number of issues to be addressed:

Implementation Structure There should be a systematic mapping between the lowest level of architectural design, and the implementation. The flexibility of the architectural style must be transformed into software, while honouring the restrictions imposed by static analysis.

Change Impact If the scope of the family changes, so that changes are being made which were not represented in the analysis, then the architectural elements may no longer provide change impact boundaries. Conventional change impact analysis would be used to assess this situation, but it must be extended to cater for the impact of, for example, requirements changes or technology changes.

[1] This should obviously support, not replace, existing rationale

The Ragged Domain Similarly, the responsibility for the provision of functionality can change from one family member to another. This forces design elements to migrate between concurrently-engineered systems, or even into and out of the embedded systems development itself. Any change impact strategy must cope with this phenomenon.

Database Support Information must be stored about the whole family of systems, with explicit indication of dependencies and variations, and in multiple artefacts. Individual instances of this information must be retrieved and validated to ensure that the links are representative of the dependencies between artefacts. The information storage tools must be able to support these linking and validation processes.

Family Evolution The family as a whole will evolve. As its members evolve, new members are added, and old members are retired. However, engine systems can remain in service for thirty years or more; While the mechanisms which control the evolution of the family must allow for these evolutionary paths, they must also cater for legacy family members.

7 Summary

We have derived a family-based embedded systems development scheme which reflects the commonality and variability among the family members, from existing research in the field. A domain-specific architectural style has been added in order to introduce the representation of appropriate non-functional properties at high levels of abstraction. These methods are being used in a large case study in collaboration with Rolls-Royce plc., to determine their effectiveness in the engine controller domain.

Acknowledgements. Much of the work presented in this paper has been conducted in cooperation with Rolls-Royce plc., under the EPSRC-funded CONVERSE (GR/L42872) project. CONVERSE is a part of the Systems Engineering for Business Process Change (SEBPC) managed research programme. We would like to express our thanks to the EPSRC for funding this research.

Rolls-Royce plc. supports a number of University Technology Centres (UTCs), with which they conduct research into key aspects of engine development and design. A UTC in Systems and Software Engineering was founded in York in 1993, and is currently working to support the case study work presented in this paper.

References

[1] A. Burns and A. Wellings. *Hard Real-Time HOOD: A Structured Design Method for Hard Real-Time Ada Systems.* Elsevier, 1995.

[2] J. Coplien, D. Hoffman, and D. Weiss. Commonality and Variability in Software Engineering. *IEEE Software*, 15(6):37–45, November 1998.

[3] JIMCOM. The Official Handbook of MASCOT. Technical report, Joint IECCA MUF (MASCOT User's Forum) Committee, June 1987.
[4] Mike Mannion, Barry Keepence, Hermann Kaindl, et al. Reusing Single System Requirements from Application Family Requirements. In *Proceedings of the 21st International Conference of Software Engineering*, pages 453–463, May 1999.
[5] H. R. Simpson. The MASCOT Method. *Software Engineering Journal*, 1(3):103–120, 1986.
[6] A. Vickers, P. Tongue, and J. Smith. Complexity and its Management in Requirements Engineering. INCOSE UK Annual Symposium — Getting to Grips with Complexity, Coventry, UK, 1996.

Stakeholder-Centric Assessment of Product Family Architecture

Practical Guidelines for Information System Interoperability and Extensibility

Tom Dolan[1,2], Ruud Weterings[1], and Prof. J.C. Wortmann[2]

[1] Philips Medical Systems Nederland BV, PO Box 10.000, 5680 DA Best, The Netherlands
[2] Technical University, Department Information and Technology, PO Box 513, 5600 MB, Eindhoven, The Netherlands
tom.dolan@philips.com, ruud.weterings@philips.com

Abstract: This paper presents a method for software product family teams to assess their family architecture with respect to family-relevant system-qualities. The method extends current architecture assessment practice through its explicit orientation to family-issues; its emphasis on those family-stakeholders neglected by conventional development methods; and its focus on providing practical "how-to" guidelines and mechanisms to enable method-users to successfully complete the required activities. Initial design and implementation of the method addresses the important system-qualities of interoperability and extensibility. The paper has a practical orientation, and concentrates on illustrating research results using industry-based examples from case-studies where the method was applied in the ongoing architecture development of commercial medical information system product families.

Introduction

Product[1] families and their associated product family architectures are created and maintained by collaborating, multi-disciplined teams. Such teams are the most effective place to instil the demand-for, and practice-of architecture assessments as a means to improve and sustain architectural quality. The most established architecture assessment method is the SAAM, and its successor ATAM, from the Software Engineering Institute (SEI) - which describes a general process for *third-party* (to the development team) driven architecture assessment. SAAM has been extensively reported, especially in its application to the system-qualities of modifiability, performance, portability.

This paper aims to contribute to the increased adoption and effectiveness of SAAM-based architecture assessment in the industrial community by:

- providing an initial toolkit of "*how-to*" techniques to enable development teams (themselves) to complete the various activities and artifacts prescribed by SAAM;

[1] Throughout this paper - the "*product* families" in question are information system families.

F. van der Linden (Ed.): IW-SAPF-3, LNCS 1951, pp. 225-243, 2000.

- incorporating the family-concept into the method - in particular seeking to apply architecture assessment to the key product-family qualities of interoperability and extensibility.

The ideas will be presented in the context of a SAAM-like method developed through industrial experience with product family creation and maintenance in the medical information system domain. The focus of this workshop paper is on highlighting the practical-aspects of the method by describing some of the toolkit of techniques to help the assessment team complete the activities in the method, particularly those related to specifying and communicating stakeholder requirements.

Architecture Assessment in Product Families

The benefits of a product family approach to software development are widely accepted in the mainstream software industry [1]. The need to manage variety and commonality (or flexibility and reuse) across family members has long been recognised [2] as a fundamental aspect of this approach. Another important aspect of product family development which has received most interest from the "product line" research effort at the SEI [3], and by the work of Ron Sanchez in "strategic product design" [4], is the long-term, strategic issues relating to the leveraging of current product design investments across other family members and indeed future generations of the family.

This *future-proof* aspect of families means that yet-to-happen changes to family members and/or the application domain must be supported by the product. The situation is exacerbated in the information systems industry where products will contain large amounts of customisation; even after installation as customers use standard technology to provide independent enhancements, which must be sustained throughout family-evolution.

The ability of software system families to provide the flexibility needed to accommodate the various changes they undergo during their life-cycles is heavily influenced by the *system-qualities* [5] (also termed "emergent properties" [6]) such as: extensibility, interoperability, reusability, scalability, portability. These qualities are all key competitive aspects of information system family business.

Architecture has been widely cited as an integral part of successful family development [7]. The architecture contains the earliest, most important, and most long-lasting system decisions. These decisions guide development and have a determining influence[2] on the family-critical system qualities mentioned previously. In addition to determining system qualities - the architecture is also used as forum for shared understanding, consensus and communication among the family development team, and is the earliest point in the system life-cycle when the system can be analysed [1] to answer "what-if?" questions. This latter point has encouraged the assessment of architecture as a means to ensure that early-design is aligned with strategic family objectives. The collaborative, interactive nature of family-development, coupled with its strategic importance for the business means that architecture-based communication and assessment should be firmly embedded in the family-development team.

---

[2] It is important to understand, however, that while a good architecture is *necessary* to ensure quality; it is not *sufficient*, as inferior downstream design and implementation can always compromise architectural design [1].

Product Family Challenges for Architecture Assessment

A **product family** is defined here as: *a collection of closely-related products (family-members), treated as a cohesive entity by stakeholders in order to leverage similarity and co-ordinate variety, both within a specific generation of the products and across generations*.

Deciding which products are *related* and why is at the stakeholders combined discretion – but the dominating factors are the similarity of the application domain addressed by the individual products (market-oriented) and the similarity of the technological resources used to develop the products (development-oriented). Families are typically manifested in one or more combinations of the following situations [7]:

- a suite of applications which work together to satisfy related business-processes e.g. MS office, banking systems software;
- a single application which comes in variants e.g. Ericsson AXE switching system - with different variants per country and per size (e.g. small, medium, large);
- independent applications built from a common component base e.g. MS Foundation Classes.

In practice families are rarely "green-field" developments – especially in the application suite approach - previous single-product efforts have established the market and experience necessary to initiate a family approach. This in turn means that existing separate-systems must be grafted (sometimes with force!) together to realise the future family, and also that an installed-base exists which demands migration into the family-concept. The fact that product family definition straddles both the past and the future, for multiple, (loosely-)associated products adds to the complexity of the family-management function.

The importance of system-qualities for product family development, and the need for a long-term, multi-product business vision from stakeholders also introduces challenges at the tactical and operational levels of software development. The bulk of effort in development methodologies and tools, however, has focused on non-family, functionality-driven products, processes and stakeholders [8]. Family stakeholders, therefore, are not well supported with practical development processes and tools necessary for family development.

Providing support in this area is the central topic of this paper. In particular, those family-stakeholders concerned with developing families as a suite of co-operating applications need support engaging in *explicit* communication of requirements concerning the important qualities of interoperability and extensibility with architects. The suite-approach is one of the most common and natural strategies among established firms in defining a family; and these qualities are essential in ensuring the smooth interaction and continuity between the various family-members expected by the market.

The SAAM Architecture Assessment

Arguably the most popular architecture assessment method is the SAAM/ATAM method. This method is officially backed by the SEI as an integral part of their developing Software Architecture Initiative strategy [3]. In addition to being non-proprietary, the method is undergoing active extension and adoption by third-party

researchers. It is therefore likely that the method will become widespread and perhaps eventually standardised by the SEI. The main steps in the method are presented in Figure 1 below.

The SAAM (Software Architecture Analysis Method) [8] is a stakeholder-centred, use-case based assessment method intended to analyse architectures with respect to various non-functional system-qualities. SAAM has been described and reported on extensively ([9], [5], [10]). The continuing progress and penetration of SAAM makes it a solid basis on which to found ongoing assessment improvement - such improvements are addressed in the method proposed in this paper.

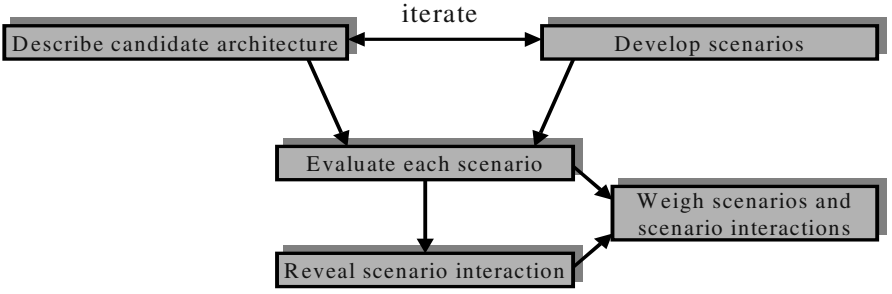


Figure 1. overview of the SAAM method

SAAM - Extension Opportunities for Family Development

The assessment method presented herein draws much from SAAM; and aims to extend it in some important directions relevant to product family development. In particular the following aspects of SAAM provide opportunities for enhancement:

- SAAM, is not particularly focused on families of systems (although does not exclude them), and most reported experience is on single-system architectures, and addresses the roles of operational-stakeholders from the user or development communities. SAAM has also focused on a subset of the important system qualities *viz.* security, adaptability, portability. Experience of family-oriented assessment, particularly in relation to other qualities (interoperability, extensibility) is a useful contribution to the general architecture assessment experience-base.
- SAAM needs use-cases to be generated (step 2 of the method) – but does not provide any guidelines as to *how* they might be identified, generated, represented – this is particularly important for system-qualities as there is no broad experience/method base from which the stakeholder can draw upon. Providing guidelines and techniques in this area contributes towards increasing the adoption/usability of the method in industry.
- SAAM was originally intended to compare candidate architectures – but is also very applicable towards establishing increased stakeholder understanding of a single architecture. It can be used early in the requirements analysis phase of development to assess the impact of requirements (so-called "discovery-reviews" as described in [10]) therefore providing a valuable tool in requirements negotiation and in family definition. This more intensive stakeholder involvement is particularly appealing for product family architectures.

The argument here is that *internally-driven* assessments can be used in addition to the external-oriented assessments typified by SAAM, and are necessary in order to embed architecture-based development in the organisation in the context of incremental architectural quality improvement. Further, these assessments can be used to provide useful material for the more formal external assessments, thus making these latter assessments more efficient by allocating the preparation-tasks to the development team, and letting the external assessors concentrate on assessment tasks.

The research contribution reported here aims to build on the SAAM and extend it with:

- a focus on the **family-development-team** helping itself as regards assuring the quality of the family architecture, rather than depending *exclusively* on external architecture consultants to police the process;
- practical **tools/tips** and processes to support more *formalised* stakeholder-architect interaction in "how-to" implement the assessment; especially as regards requirements *identification*, *specification* and architecture communication.

The overall approach to this product-family and "quality-specific" extension to the basic SAAM method is summarised in the Figure 2 below. It recognises that useful advice regarding implementation of quality assessment is dependent on the application domain and the particular qualities involved. The approach here is to develop domain and quality "*add-ons*" (see middle-, and top-layer of Figure 2) which integrate with the general process and extend it to provide practical tools/techniques[3] to fulfil the steps. The extensions described focus on:

- tailoring the general process for the domain of information system families;
- adding practical tools to support participants in generating the necessary process artifacts for the qualities of interoperability and extensibility in the domain.

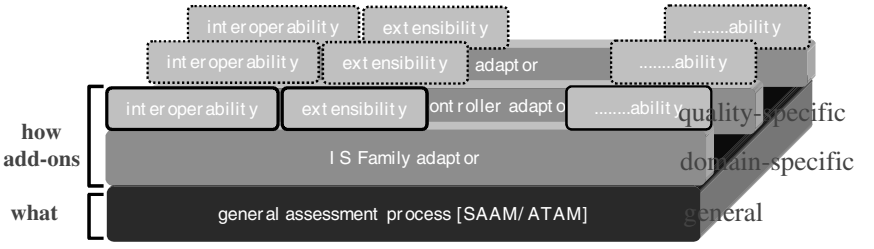


Figure 2. add-on based architecture-assessment improvement approach

These are initial steps on the road to providing a comprehensive set of domain-, and quality-specific architecture assessment tools. The long-term aim is that through increased industrial and academic exposure in these (interoperability and extensibility for IS families) and other domains/qualities; those domain-/quality-specific parts will mature to plug-in library process-components on top of a general assessment method. This depends heavily on adoption and experience-reporting by the practising community.

[3] These tools comprise a set standard-questions, templates, process-techniques which, after repeated exposure and refinement, will evolve into a good-practice process handbook; similar to those in use in more mature industries e.g. chemical, building.

As extensions are provided by others (dashed-add-ons in Figure 2) – new possibilities for combining domain-extensions may emerge, and as the general understanding of the system-qualities increases, it may be possible to derive domain-independent tools.

An assessment method addressing these concerns is presented in the next section, illustrating usage of the tools with reference to industry-based interoperability and extensibility issues in medical information system families.

Method: Family and Quality Oriented Implementation Tools

This section briefly presents the overall design of the assessment method (see Figure 3). The process steps are motivated by SAAM [9], SEI report on architecture assessment best practice [10], ATAM [11]. Due to space-constraints, detailed description of the process steps (roles, inputs, outputs) is *not* described here, but are fully reported in a pending PhD thesis by one of the authors; a graphical example of the process description for step 1 is included for illustration only. The focus of this workshop-paper is on describing the extensions developed to support the application of the method to information system families and the qualities of interoperability and extensibility. Most attention will be devoted to describing the toolkit of techniques and tips developed to help stakeholders and architects explicitly communicate family quality requirements, especially in the setup and information-gathering phases of the method.

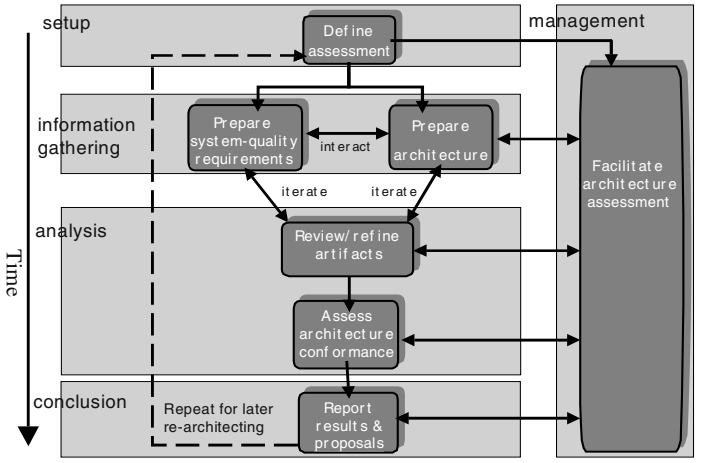


Figure 3. Family Architecture Assessment Method - Overview

Define Assessment

Goal: to define the agreed scope and intention of the assessment with the family stakeholders, and to plan pre-, post-, and actual-assessment .

Toolkit/Tips

- Seek to identify two to five clear goals related to system-quality (i.e. in this research interoperability, extensibility) for the assessment - do not try to determine if the development project will meet all targets;
- Establish whether the focus is requirements-discussion "architecture discovery" type review; or a formal "architecture evaluation" against a fixed set of requirements - (see [10]);
- Establish the purpose and audience of the assessment report and what will be done with it - it is important to get management commitment and participant understanding that the evaluation is improvement-oriented and not the basis of stakeholder/architect performance-appraisal;
- Participants:
  - the four primary family stakeholders are typically [8]: family business manager, product manager, customer support manager, and development manager; or their representatives;
  - The architect (or architecture team);
  - The assessment facilitator (to guide the assessment);
  - Application domain expert (to provide consultancy on application domain;
  - External architecture expertise (optional) usual for a more formal evaluation;
  - Administrative/logistical support (optional).
- Identify the stakeholders and indicate their roles, interests and associated assessment documents;
- Gather available requirements documentation related to the system-qualities under consideration;
- Use the ***change-case-guidelines***, ***family-feature-map***, ***system-topology*** and ***migration-map*** to identify candidate *change-cases* for interoperability and extensibility;
- Get an architecture description(s) - indicating computation and data components, and all component relationships (connectors) [SAAM-method step 1];
- Typically for family-related issues the *conceptual* and *deployment* views from the ***4+1*** model [12] are important;

Sample Process Overview

Figure 4 below shows a sample graphical representation of the process description developed in the research. Such representations summarise the detailed activities associated with each step in the method; emphasising the: nature (group/individual) of the step, participants involved, sequence of activities, the tools used, and the inputs (left-side) and outputs (right-side).

Prepare System-Quality Requirements

Goal: to establish a pre-assessment idea of the overall status of the requirements with respect to the assessment goals and identify/specify the change-cases with the stakeholders.

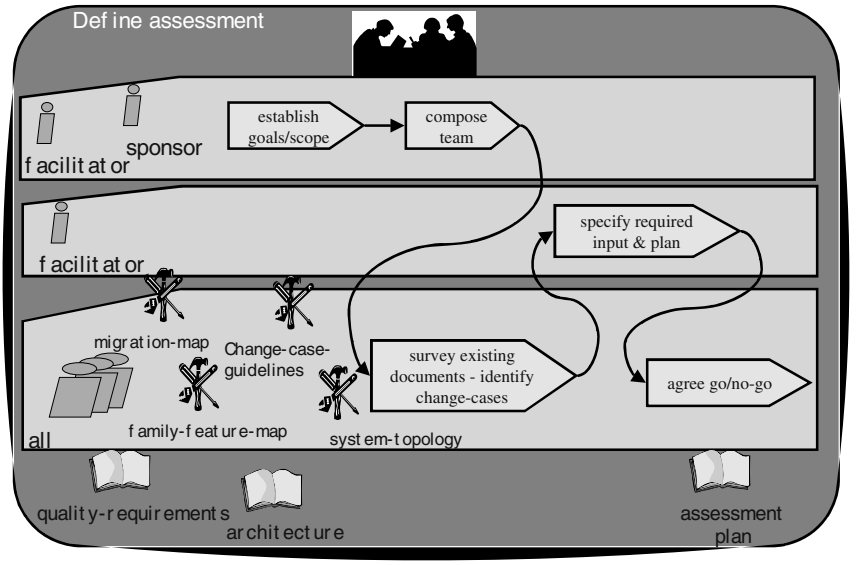


Figure 4. Example of graphical process description for method step

Toolkit/Tips

- Establish a definition of the family - if it does not exist. The ***family-feature-map*** concept described in section 0; provides a useful overview vehicle to help elicit the context, scope and variability within the family.
- Use the ***change-case guidelines*** to concentrate the stakeholders effort on the most important 5 - 7 change-cases to be dealt with per assessment cycle.
- Specification of *change-cases* for interoperability and extensibility is typically text-based and is structured for assessment use by a ***change-case template*** structure as presented in section 0.
- Rank the change-cases in order of priority with the stakeholders as a group using general ***ranking criteria***.

A Note on "Change-Cases"

The term ***change-case*** (see [13], pp63) is used to describe those candidate changes that must be accommodated by the system. They play the same role in representing requirements for the "build-time" aspects of a system, as the use-cases play in describing the functional requirements associated with the "run-time" operation of the system. The new term distinguishes these two aspects of the system life-cycle, and emphasises the fact that change-cases specifically deal with issues of most interest to the system producer in particular the future modifications to the system and the system's relationship with others - which are critical in family-definition.

Prepare Architecture

Goal: to establish a pre-assessment idea of the overall status of the architecture with respect to the assessment goals and identify/specify the representations, rationale necessary for the assessment

Toolkit/Tips:

- Describe the architecture using existing representations in the organisation with an emphasis on the computational components, data components and their relationships. Existing views shall be embellished (where needed) with the information defined in Kruchten's ***4+1*** views [12].
- Typically for the chosen system-quality issues in information-system families the following views have proven most useful:
  - logical view (information model) – static issues
  - deployment view (software modules in development organisation) – static issues
  - process view (e.g. UML Interaction diagram) - dynamic issues
  - use-case view (illustrate, in stakeholder terms, how the elements work together) – dynamic issues
- Provide evolving change-cases to architect so that the architecture representation is driven by requirements.
- The architect should also indicate which changes-cases are related to which views – this is useful to prioritise architecture views.
- The precise views, their granularity, and completeness is the responsibility of the assessment-team[4] – if the stakeholders are satisfied then it's sufficient. The approach in this method is that it enables professionals to carry out an assessment; the assessment-team is responsible for its own actions. Of course downstream development will have higher expectations of architecture completeness, but that is outside the scope of this assessment method.

In steps 2 and 3 above there is typically (facilitated) interaction, but not architecture assessment between the stakeholders and architect.

Review/Refine Artifacts

Goal: to review the material gathered in the specification-phase and to develop a technical and organisational assessment-execution plan.

Toolkit/Tips:

- Verify that the basic requirements/architecture specifications/views are available – check against ***change-case template*** ,***4+1***.
- Use general assessment ***requirements ranking criteria*** to establish priority of requirements. Sample ***change-case-guidelines*** for specific interoperability/extensibility qualities can be used to further refine the ranking - so that the most important change-cases are carried further.
- Related (in the domain or architecture) and/or conflicting change-cases are particularly important and their dependence should be reflected in the ranking and criteria setting.

[4] More focused guidelines towards identifying the appropriate architectural views and granularity shall be developed with increased exposure to ongoing industrial application

Assess Architecture Conformance

Goal: To assess the conformance of the architecture to the quality-requirements as specified by the stakeholders.

Toolkit/Tips:

- The ***assessment criteria*** may be used to structure the discussion and offer opportunities for brainstorming, creative dialogue towards compromise
- Architect uses functional use-cases or *direct* (no modification needed in architecture - see SAAM) change-cases to explain architecture and show how it supports requirements.
- For *indirect* (modification needed to architecture) change-cases - the various possibilities and costs of changes must be discussed with stakeholders [SAAM method step 3]. This is the discourse intended to help architect and stakeholders have a structured exchange on architectural/requirements issues and to come to a shared understanding of issues and (later) solutions.
- A tabular overview (***requirement-architecture interaction matrix***) listing change-cases, affected components and associated costs (including negative side-effects) is useful here for overview [SAAM method step 4], and to illustrate the mapping between requirements and architectural domains.
- Architectural elements which are affected by un-related change-cases (*trade-off points* in ATAM) *may* indicate a weak separation of concerns (this may be expressed in an ***interaction metric*** totalling the change-cases related to each component. It may just indicate a very important component – again individual context is determining here.
- Overall evaluation - weigh each *indirect* use-case and the use-case/component ***interaction metric*** in terms of their relative importance. This process is used to establish the overall performance of the architecture, and should be done by all affected stakeholders to achieve consensus. [SAAM method step 5]. The pre-defined ***assessment criteria*** is useful here.
- The issues, scores, important discoveries raised during the review should be recorded in minute-form by the facilitator (or appropriate administrative support) in the ***assessment-log*** of the assessment and is will be input to the official ***conformance statement***.

Report Results and Proposals

Goal: To document and report the assessment results to participants and other concerned parties; and to establish a baseline for follow-up assessment or architecture/requirements rework.

Toolkit/Tips:

- A summary of the architecture's accommodation of the required interoperability/extensibility shall be provided by a **conformance statement**. This shall be provided by the facilitator (with the co-operation of the architect) and is intended to communicate the assessment findings to the stakeholders and management. The ***requirements-architecture interaction matrix*** discovered during the assessment-meeting will be important here in illustrating any system quality limitations of the architecture.
- The weighted use-cases and interactions should be used as the basis of planning any *re-architecting* activities in order to improve architectural support for the indirect use-cases. Re-architecting should only be considered in conjunction with stakeholder feedback. Important in analysing any changes is that they do not adversely affect any *direct* change-cases or other system functions/qualities - this is the architects responsibility.
- A set of proposals - including repeat assessments (see dashed-arrow in Figure 3) for modified/ignored requirements or architecture elements - should be proposed with rationale and risk-assesment for incorporation into ongoing family-planning discussions with stakeholders.

Facilitate Architecture Assessment

Goal: To co-ordinate the participants and ensure locations, materials necessary for efficient working. The facilitator also provides support in explaining, guiding and reporting the method.

Toolkit/Tips:

- Ensure organisation management supports process and receives assessment report
- Provide example-based introduction to method and ensure buy-in from team before starting

An overview of individual tools and their traceability to the above process steps is provided in the following section

Assessment Method Tools

This section describes a representative-subset (shaded grey in Table 1)of the most important "how-to" techniques borrowed/developed to aid method implementation, as presented above. These represent the initial contributions of the research towards extending SAAM and assisting stakeholders to provide the artifacts necessary for architecture assessment. Techniques will be illustrated with material from a number of industrial case-studies from the medical information systems domain, where the method is undergoing industrial-trials.

In order to protect confidential information and to make the material accessible, simplification and alteration of details has occurred.

Table 1. Overview of technique usage in method steps

Tools\Method-steps	Define	Req.	Arch	Review	Assess	Report	Mgt.
Family-Feature-map	✓	✓	✓	✓			✓
System-topology	✓	✓	✓	✓			✓
Change-case-template		✓		✓			
Migration-map	✓	✓	✓	✓			✓
Change-case-guidelines	✓	✓		✓			✓
Requirements ranking criteria		✓		✓	✓		✓
4+1			✓	✓	✓	✓	✓
Assessment criteria	✓			✓	✓	✓	
Req-arch interaction matrix					✓	✓	
Interaction metric					✓	✓	
Conformance statement						✓	

Family-Feature-Map

Definition

The ***family-feature-map*** is a tabular structure listing which features are contained in which family-members (variants). Further it indicates whether these (mainly functional) features are a *core* part of the variant (i.e. always present/enabled, and standard included in the variant); or whether they are *optional* (can be enabled depending on customer configuration). This can be developed using e.g. domain modelling, or the commonality analysis techniques advocated by [2].

Role in Assessment

- This is a concrete *definition* of the family in business/application-terms, and provides participants the vocabulary with which to discuss the commonality, variability, boundaries and internals of the family members.
- Practical experience shows that the exercise of defining the ***family-feature-map*** will stimulate the stakeholders to consider and generate change-cases.
- The absence of (or ability to generate) this artifact is an early-warning that the family is not well established - a meaningful (i.e. realistic) family-architecture cannot exist without a solid business-case for the family as reflected in the ***family-feature-map***.

Who

Typically the ***family business manager*** is the stakeholder primarily responsible for defining the content of the family in response to application and market requirements. Other family stakeholders and (even) the architects may be involved as part of the collaborative family definition process. Individual ***product managers*** (commercially responsible for individual variants) use the ***family-function-map*** to co-ordinate their activities and limit their area of responsibilities in subsequent roll-out of the members.

Example

Medical IS Family-members

Features	Entry	Standard	Enterprise
Pat-reg	O	O	C
Request-reg	C	C	C
Appt.-sched.	C	C	C
Exam-admin	O	O	C
WLH	C	C	C
reporting	O	O	C
Mgt.-reporting	-	-	C
Review-prep	-	C	O
viewing	O	C	O
Archive-mgt-F	O	O	C
Archive-mgt-D	-	C	O

C - common feature
O - optional feature

Figure 5: Sample family-function-map showing core/optional function-elements in each of 3 family-members

Limitations

The family-map does not show the variability *within* a particular functional element e.g. the different types of viewing functions supported (but this could be remedied by simply increasing the amount of detail shown). It also does not show dependencies (either commercial or technical) between optional-functions. Such additional detail can prove necessary in the assessment – this is typically indicated by the architect in the iteration between the architecture- and requirements-preparation steps in the method.

In other domains familiar to the authors, architectures are (in some cases beneficially) over-designed to satisfy all-combinations of features - thus allowing un-planned feature configurations in the future. While this is *good*-architecting (accommodating the unknown) it should not be an excuse to ignore stable domain-commonality and not to consider leveraging this in design. Encouraging (forcing!) commercial stakeholders to make informed choices based on domain realities is good-practice even if later the architect will not "boiler-plate" all aspects into his design.

System Topology

Definition

The ***system-topology*** is a simple graphical-overview of the various systems/components which are (or can-be) present with the family in its operating environment.

Role

The ***system-topology*** is used primarily as a means to:

- provide a real-life product-based representation of the current family content and boundary in its application context
- a means to provide insight on the actual and candidate systems that *interoperate* with the family

- a means to indicate possible systems which could be incorporated/replaced by the family thus *extending* the family-scope.
- Act as a means to encourage stakeholder thinking on **interoperability** requirements by describing the candidate "actors"; and on **extensibility** opportunities by highlighting associated applications in the domain. This prompts stakeholders to generate interoperability and extensibility change-cases.

Who

Typically the ***product manager*** and/or the ***architect*** provide the context in which the family or individual products therein operate.

Example

The example in Figure 6 below illustrates the various internal (based on the suite-concept of family indicated in section 0) and external components that provide the application context for the PACS (Picture Archiving and Communication System) image management system. The *ultrasound modality* highlighted as a variant of external system in the figure will be referred to later in this section when illustrating an example interoperability change-case.

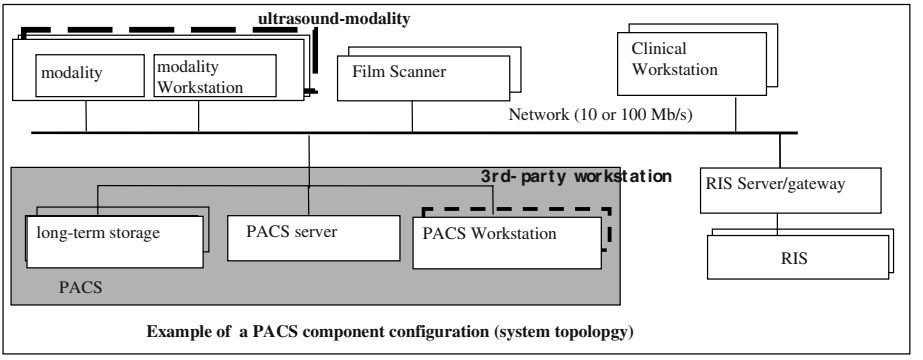


Figure 6. Sample system topology showing the internal and external components required to interoperate in the (*PACS*) family application domain

Limitations

The ***system-topology*** provides a static view of the systems in the application domain, it contains no information as to *how* interaction occurs dynamically, or specifies data/control interfaces. In early-design architecture assessments this is not a limiting factor, for stakeholders - architects typically will need such information to have a convincing story at assessment-time.

Migration-Map

Definition

The ***migration-map*** is a simple graphic illustrating the various members of the proposed family and their ancestors and planned-descendants over time. This tool captures the past/future aspects of family indicated as a challenge in section 0 - analogous to a family-tree in genealogy.

Role

The ***migration-map*** sets out the roadmap for the family - it is especially useful in encouraging business stakeholders to make the following important (especially for information system families) decisions regarding family continuity and **extensibility** explicit:

- What legacy systems must we migrate from and which (new) members are affected
- What level/type of migration is required above - examples
- What individual family-members are contained in the family and how/which members do we offer customers the ability to move/extend to within the family

Further it provides motivation for stakeholders and architects to pre-empt the risks and issues associated with legacy migration in the family - a very major issue in information systems community.

Who

Typically the ***family business manager***, ***customer support manager*** are responsible for defining the bridge between past and future to be supported by the family. The ***development manager/architect*** also plays a leading role here because of the need to account for the costs of technical discontinuities associated with trying to please all the existing-customers and all the future-customers. This is typically an area where commercial wishes and technical limits conflict - it is vital that the discussion is clarified early in the family's life - thus increasing the chances of business and technical alignment.

Example

The example below is based on a *ficticious* product family (RIIMS) which integrates previously separate information- and image-management systems. Without going into many details - the main point is that the stakeholders have indicated which members belong in which market segments (entry, standard, enterprise); which legacy migrations must be supported - and therefore which not (e.g. *legacy-RIS* to *RIIMS* but not to *Basic RIIMS*). The diagram also captures the strategy which dictates which upgrades/extensions *within* the proposed family the producer will support - in many cases this is strongly motivated by the technical differences foreseen between entry-level and enterprise systems, and provides a means to optimise revenue-streams by forcing customers to follow a planned-path of expenditure, and reducing the amount of technical-gap between extensions.

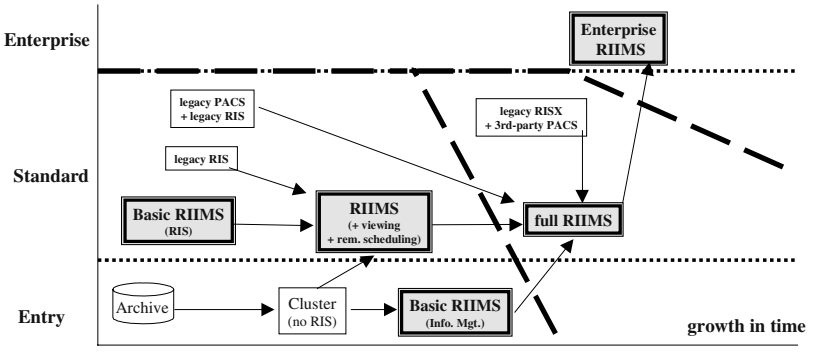


Figure 7. Sample migration-map showing the existing and planned family-members and the required migration/extension between them

Limitations
The ***migration-map*** does not show the variability *within* a particular member e.g. the features supported - but this can be derived from the appropriate ***family-feature-map***. It forces technical considerations to the business-level - but there is no guarantee (as with other tools) that things may not change.

Change-Case Template

Definition
The change cases are the primary means to represent requirements in the method, and the template is used to structure the change-case. The template provided by the method will be described below with an example
Role
The template aids the stakeholder in expressing interoperability/extensibility requirements in a form suitable for assessment.
Who
The ***stakeholder*** uses the template to specify the change-cases and the ***architect*** uses it to extract relevant information.
Template
Change-case Name: Name of change-case and associated system-quality.
Goal: the intention of the change-case.
Actor: the stakeholders (person/other-system) affected by the change.
Trigger: The stimulus for the change (i.e. who, what initiates the change).
Brief Description: a brief description of the stakeholder-system interaction that arises as a result of the change of the change.
Rationale: Short description of the business case (preferable with reference to requirements, strategy documents) motivating the change – if different from "trigger".
System-quality Context: This is context in which the change-case occurs. The environment both in terms of the system (the family-member) and its surroundings should be described as much as relevant. In particular:

- System topology (i.e. relationship with other systems, users)
- System settings (any special configurations, the particular family-members involved)
- Pre-conditions (necessary resources or conditions needed before the change can occur)

System-quality Details: Any more detailed system behaviour required to support the behaviour.
End note: any notable conditions, circumstances for the system stakeholder after the case has been implemented. In particular if any other change-cases are affected (triggered

Example - This is an interoperability example based on the system topology diagram previously

Change Case # 1

Provide handling of UltraSound(US) Images and Measurement data by brand-X PACS

Actor: UltraSound Modality

Trigger: US Images and related measurement data from the US modality – they shall be accommodated

Rationale:
More and more US modality vendors are confronted with (users) requirements to offer PACS based solutions in their product portfolio, optimized for US (color) images and other measurement data. There a two business opportunities for the brand-X PACS in this domain;

- a dedicated US Image Server/Archive for the (larger) UltraSound lab
- a versatile Radiology PACS, supporting multi-modality imaging devices; CT, MR, XA, US.

Brief Description:
Brand-X PACS (hereafter called the PACS system) should be able to receive, store, process and display (color) image data and related measurements (results) from the US modalities.US studies may consist of US (color) images, both ***single frame*** and ***multiple frame*** series. These images are transferred in a lossless of lossy (RLE) type of compression mode, dependant of the type of study, and may be accompanied with US measurement results, as well as ECG (curve) data.
The PACS system should be able to handle and display these (color) images and measurements, so that the (primary) user of the system can review and manipulate these images on the systems (diagnostic) viewstation. Dedicated US practitioners may want to use a color-based viewstations and combine these images with other type of examinations (e.g. CT or MR); other radiologist may want to use their high-resolution (B/W) diagnostic viewstation instead.
In addition, US examiners may want to review the dynamic behavior of the US (multiple frame) images in a 'cine' loop type display; simultaneously looking at the moving ECG (curve).

Interoperability Context:
The system shall support US images according the ***DICOM standard***; both single frame and multiple frames (cine-view) are to be supported. Large US cases may be transferred in a ***lossy-compressed*** mode, using the ***RLE compression method***.
A typical US examination case consists of patient/examination information, a set of US Images, measurement results, ECG data/curves (the latter in case of cardiac studies) and an audio (speech) file. The PACS system should accept these types of images and multi-media information as a complete 'acquired image' set and stores all these images and information objects accordingly.
The PACS system should allow dedicated (color) viewstations to be optimized for reviewing these US studies, making use of all the US (image) object elements as described above (inclusive speech replay).
In essence, all US images and other information are shown on these viewing stations 'as last seen' on the US modality.
Moreover, these US viewstations should be able to select one or more US image out of a examination series, add patient demographics and measurements data to it and to store the resultant image, called ***photofile image*** into the systems database for later reference or clinical reporting or teaching purposes.
These photofile images have then become part of the patient's (history) folder and it should be able to make a ***video hardcopy*** and/or export the photofile image(s) to an external DICOM workstation.
Other PACS-based (B/W) workstations may present these US images 'as best as possible'; restrictions on the US image presentation, image manipulations and the speech replay function may apply though.

End note
Assuming the US requirements as stated above are implemented in the system, another Change Case may be considered to facilitate the system for dedicated ***IVUS (IntraVascular UltraSound)*** studies.
IVUS studies are specialized examination procedures, performed on a Vascular X-ray modality with an add-on US acquisition device. A typical IVUS study consists of a series of XA images and US images that are correlated in the spacedomain (i.e. the X, Y, and Z coordinates of the two image planes are transferred with the images). Based on these (XA+US) image data sets, a three-dimensional, computerized graph of the vessel structure can be reproduced.
The PACS system should be prepared to support the storage of these IVUS examinations, and accommodate the reproduction and display of the vessel structures on the US (color) viewstations.

Conclusions and Future Work

In this paper the case for stakeholder-driven self-assessment of family architecture by the family team has been made, and the subsequent need for "how-to" extensions to the basic ideas and mechanisms promoted by SAAM/ATAM indicated. An outline of such a method has been presented initially concentrating on interoperability and extensibility of information system families. A framework for extending this method to other domains and qualities has also been suggested. Specific attention has been given to advice and tools needed to implement the method steps and a selection of these tools has been described with specific attention given to explaining their role in the method through practical illustration in ongoing architectural assessments of medical information system families.

The authors experience is that family stakeholders and architects are very receptive to the mechanisms which help them get some concrete (documented) input on the often vague-business of family definition and maintenance. The intention is that other researchers interested in practical applications of family-management will be motivated to experiment with the initial tools and framework presented herein with a view to critiquing/enhancing the approach.

Continuing work (and publications) by the authors will concentrate on:

- Reporting complete case-studies of experience with the method - especially positive/negative experiences or insights, and exposing the method in other domains
- Communicating other aspects of the method, mentioned in passing in this workshop paper:
  - describing remaining toolkit components (e.g. change-case selection guidelines; ranking criteria)
  - providing more detailed description of the individual steps in the method
  - advice on which architectural views and granularity needed for interoperability and extensibility;
- increasing the application of quantitative metrics in architecture assessment;
- measuring the effectiveness/contribution of the method

References

[1] Bass, L, P. Clements, R. Kazman; *Software Architectures in Practice*; Addison Wesley Longman; 1998; ISBN 0-201-19930-0.

[2] Weiss, D.; “Commonality Analysis: A Systematic Process for Defining Families", *Development and Evolution of Software Architectures for Product Families - Proceedings Second International ESPRIT ARES Workshop, Las Palmas de Gran Canaria, Spain, Feb. 26-27*; Frank van der Linden (Ed.); Springer-Verlag; Berlin 1998. ISBN 3-540-64916-6

[3] http://www.sei.cmu.edu/plp/index.html

[4] Sanchez, Ron.; “Strategic Product Creation: Managing New Interactions of Technology, Markets, and Organisations”; European Management Journal, Vol. 14 no. 2 April 1996; pp121-138; Elsevier Science Ltd.; 1996.

[5] Clements, Paul C., Len Bass, Rick Kazman, and Gregory Abowd; *Predicting Software Quality By Architecture-Level Evaluation*; Fifth International Conference on Software Quality; Austin, Texas; 1995.

[6] Simon H.A.; The Sciences of the Artificial; The MIT Press; 1981; ISBN 0-262-19193-8.

[7] Jacobsen, I., M.Griss, P.Jonsson; *Software Reuse – Architecture, Process, and Organization for Business Success*; ACM press; New York, 1997. ISBN 0-201-92476-5.

[8] Dolan, T., R. Weterings, J.C. Wortmann; "Stakeholders in Software-system Families"; Development and Evolution of Software Architectures for Product Families - Proceedings Second International ESPRIT ARES Workshop, Las Palmas de Gran Canaria, Spain, Feb. 26-27; Frank van der Linden (Ed.); Springer-Verlag; Berlin 1998. ISBN 3-540-64916-6

[9] Kazman, R.,G. Abowd, L. Bass, P. Clements; "Scenario-based analysis of Software Architecture"; IEEE Software; Vol. 13, No. 6; November 1996; pp 47-57.

[10] Abowd, G., L. Bass, P. Clements, R. Kazman, L. Northrop, A. Zaremski; Recommended Best Practice for Software Architecture Evaluation; CMU/SEI-96-TR-025; January 13 1997.

[11] Kazman, R. M.Klein, M.Barbacci, T. Longstaff, H. Lipson, J. Carrière; "The Architecture Tradeoff analysis Method"; Proceedings of the 4th International conference on Engineering of Complex Systems ; August 1998.

[12] Kruchten, Philippe B.; "The 4+1 View Model of Architecture"; IEEE Software; November 1995; pp. 42-50.

[13] Bennett, Douglas. W.; *Designing Hard Software : the essential tasks*; Manning Publications Co.; Greenwich

ESAPS – Engineering Software Architectures, Processes, and Platforms for System Families

Frank van der Linden[1] and Henk Obbink[2]

[1]Philips Medical Systems B.V., Veenpluis 4-6, 5684 PC Best, the Netherlands
[2]Philips Research Laboratories, Prof. Holstlaan 4, 5656 AA Eindhoven, the Netherlands
{frank.van.der.linden,henk.obbink}@philips.com

Abstract. Across Europe 21 companies and research institutions work since July 1999 together on the Development and Evolution of Software Architectures, Processes and Platforms for System Families in the ITEA project ESAPS. Based upon earlier and smaller scale experiments in ARES and PRAISE ESAPS aims to improve the state of practice in European industry with respect to the Engineering of Architectures, Processes and platforms for system families in order to achieve significant higher levels of reuse and improved system quality.

Introduction

In July 1999 a consortium of 21 companies and research institutions started a project "Engineering Software Architectures, Processes and Platforms for System Families" – ESAPS. The project is based upon earlier European experiences within software architectures and development for product families, viz. The ESPRIT projects ARES [1] and PRAISE [2]. ESAPS is set up to exploit in industrial activities the methods and techniques that are developed in the ESPRIT projects. Based upon our experience in these projects, we expect that co-operating in complementary domains much progress can be obtained by otherwise competing partners. ARES has developed advanced technology for describing and analyzing product variation. PRAISE has developed an initial system family package, including process, method and variability description techniques

The idea of a program family is not new and dates back to the seminal papers of David Parnas and Edsger Dijkstra [9], [5]. However, they do not get much attention until recently. Nowadays system-families are becoming strategic business assets. On many markets, the product's economic life is becoming shorter and shorter. For designing products we have to take into account an increasing number of user groups (so-called stakeholders) with diverging requirements. In order to survive it is required to combine and balance the need for a careful engineering approach with the need for rapid product delivery [11]. We have to handle requirements and architecture so that varying products can be derived using a common pattern – i.e. a family concept. The fundamental concept of a system-family is a domain-specific product architecture based upon a layered set of platforms. The family is constructed within a software engineering process focussed on pervasive reuse.

F. van der Linden (Ed.): IW-SAPF-3, LNCS 1951, pp. 244-252, 2000.

The ESAPS project is conceived as a four years project that has been divided into two phases of two years each. The first phase will deal with *the development of the approach and laboratory scale validation* of the individual technologies and technology integration framework. The second phase will focus on the *integration* of the individually validated technologies and *automation of the approach and industrial scale validation in various domains.*

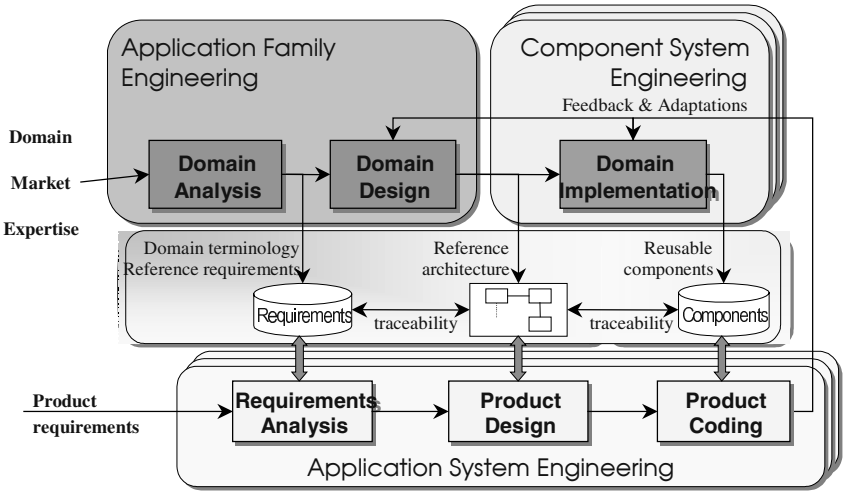


Fig. 1. Product Family Engineering organization

ESAPS Organization

The organization of ESAPS is based upon the Product Family Engineering organization, as is shown in figure 1. It combines the two fundamental different approaches to Product Family Engineering. First, Jacobson et. al. [8] determines three process categories:

1. **application family engineering** covering the development of assets usable for the complete family
2. **component system engineering** covering the developments of single (platform) components to be used within system-family members.
3. **application system engineering** covering the necessary developments to construct the family members using the components developed in 2.

The system-family methodology, resulting from the PRAISE project and earlier various DOD efforts identifies two engineering types:

1. **domain engineering** in charge of the analysis, design and the management of the domain assets. (For Reuse focus)
2. **application engineering** in charge of the development of a new product using the domain assets (With Reuse focus)

Within ESAPS we identified certain focus points to put the main emphasis upon. We have the following structure of the project:

1. Analysis and modeling of and for system-families. This work package is used within all three engineering domains, as we need analysis at all levels. However, this work package has a main focus at Application Family Engineering.
2. Definition and description of system-families. Deals with Domain Design and Component System Engineering.
3. Derivation of products and evolution of system-families. Deals with Application System Engineering.
4. Validation of the technologies developed in work packages 1-3. Applies the approaches of the other work packages in large industrial system families.
5. Dissemination. In order to maximize the synergy project-wide intensive workshops are planned to take place each ½ year.

We start with an overview of the ESAPS project's technical basis, starting with the goals. Three main subjects: Analysis, Definition and Evolution of system-families will be treated separately.

ESAPS Goals

The ESAPS project aims to provide an enhanced system-family approach and enhanced domain specific platforms for the application domains of the partners. ESAPS is designed to enable a major paradigm shift in the existing processes, methods, platforms and tools and comprises the following changes; see figure 2:

1. From state of the art *object technology* to *component technology* for complex embedded systems.
   This will allow to compose systems from available components, both in-house and COTS.
2. The transition to strategic pervasive domain specific platform based reuse. This enables the transition from opportunistic reuse to strategic reuse, based upon marketing demands instead of technical possibilities, using a platform approach.

These changes are the basis to be able to move from engineering *single systems* to the engineering of multiple systems or *system-families,* leading a large commercial diversity with a relative small technical diversity.

The ESAPS results are targeted to be fivefold:

1. Enhanced system-family engineering processes
2. Enhanced system-family engineering methods and tools
3. Enhanced component based domain specific platforms for system-families
4. Requirements for engineering tool suppliers to adapt their tools to comply with system-family engineering needs. Resulting from 1 and 2.
5. Requirements for generic and domain-specific middleware suppliers. Resulting from 3.

ESAPS Technical Basis

A system-family is defined as a group of systems sharing a common, managed set of features that satisfy core needs of a scoped domain. The idea behind a system-family approach is to build a new system or application from a common set of assets (domain model, reference architecture, components, platform) defined from earlier developed

systems belonging to the same line. A software asset is a description of a partial solution. It might be a component, known requirements or design elements that an engineer uses to build or modify a software product.

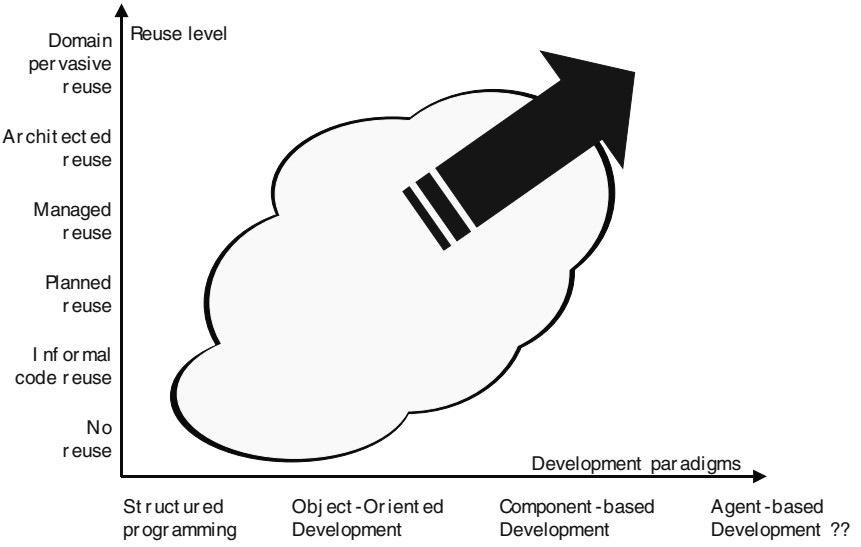


Fig. 2. Direction of ESAPS

A is well known the Business, Organization, Process & Architecture (BOPA) have mutual influences. A change in one of them will have a result in a change in at least one of the other aspects and often in all of them.

ESAPS has a focus to architecture and process. However, we are aware that we cannot deny the other aspects therefore they are investigated on a small scale within ESAPS. The business aspect deals with the issues of

- How do we earn money,
- Which kind of product do we make
- What is the scope of the domain
- Who are the stakeholders that have requirements
- Product roll-out scheme

The organization aspect deals with the issues of

- How are we organized
- What are the different roles we recognize
- What are the products that are used for communication
- What are the tools we use
- Physical distribution of people

The main concern of process aspect is:

- Who is doing what
- Who is responsible
- What do the products look like
- Acceptance process of artifacts

- Technical support

Finally the architecture aspect deals with

- The distribution of the requirements over the artifacts
- The rules that the artifacts have to obey
- The acceptance of the artifacts
- The relationship between the artifacts

In the next sections we will go into the details of the three main work packages of ESAPS.

Analysis of, and Modeling for System-Families

The first topic deals with analysis. Because of the importance of architectural decisions, they have to be carefully modeled and analyzed. If problems are found early in the software life cycle, they are easier to correct. Software quality cannot be appended late in a project; it must be inherent from the beginning.

Within Application Family Engineering the main subjects are the analysis of the problem-domain and the solution-domain. The problem domain is addressed by domain analysis and modeling, focussing on the problem domain in general. As a specific topic we deal with aspect analysis and modeling, focussing on the problem domain qualities that have to be satisfied. We consider both scenario based and model based analysis techniques. Scenario based techniques are particularly suitable for reviews while modeling based techniques are better at predicting quantitative estimates on quality attributes.

The solution domain is addressed by Architectural analysis and modeling, focussing on the development artifacts and the solution domain. Note that the latter topic also is connected to Component System Engineering and Application System Engineering via the analysis of the produced platforms, components and systems. Reverse architecting techniques can be used to assess the value of a legacy system, in checking whether the current development follows the architectural rules or in understanding the suitability of candidates when acquiring a large software system.

The domain is the abstract space where the families live. The platform is the collection of all components and interfaces that are used in all, or most, products in the family. Within domain analysis, methods have to be developed for identifying and modeling those assets that are common to all family members, and those assets that belong to specific family members only.

Besides the functional aspects of a system family there are also non-functional attributes or quality aspects of the systems to be developed. The book [3] distinguishes between runtime discernible, not run time discernible, inherent and business qualities. ESAPS uses a similar but slightly adapted approach. As can already be deducted from the different quality classes, the different qualities serve the needs of distinct stakeholders. Each of these qualities and interfaces are often very important from the view of a particular stakeholder.

Specific analysis and design techniques have to be developed and integrated to deal with the different qualities. Solutions may lie in (run-time, or development-time) infrastructure support, in the design of specific interfaces, in the design of specific services or a combination of all. Moreover, the qualities may be connected to specific architecture views [6],[10].

Definition and Description of System-Families

Designing system families requires a way of designing the commonality and variability in such a way that during the product engineering phase variants can be implemented efficiently. The system family is captured in the system family architecture, which in turn is based upon a reference architecture. The reference architecture comprises the main architectural information of the complete family. It is an intermediate between the problem space, encapsulated in the domain model, and the solution space. The reference architecture represents the earliest design decisions that are the most difficult to change, the most critical to get right and addresses most of the aspect issues such as performance, reliability, modifiability and security. Therefore the reference architecture is an important system-family artifact.

It defines the components (mandatory, optional, alternative), component interrelationships, constraints, and guidelines for use and evolution in building systems in the system family. Consequently, the reference architecture must support common capabilities identified in the specification, the commonality, and the potential variability within the system family. The system family scoping, defined by the business aspect, is essential for the development of a reference architecture since it defines the bounds for systems that will constitute the system family as well as the goals to be achieved and targeted by system family development.

The system family architecture implements the reference architecture by providing specific technical solutions. The platform and components are important artifacts in a system family architecture based upon a reference architecture. They are part of the solution space. They use heterogeneous solution technology (as, e.g., CORBA, DCOM, Beans) using the necessary integration means (middle-ware, orbs, bridge technology, wrapping techniques). In this context, middle-ware provides the infrastructure for high level co-operation between components including information exchange.

In order to support the architecture definition process we need to investigate system family oriented component development techniques addressing requirements engineering, design engineering, component management, integration testing and component evolution. In particular methods for semantic interface descriptions and related by contract based development approach will be provided.

Within this work package we also deal with the Component System Engineering, involving the platform implementation. An important task is to investigate ways to create and validate platform components.

Derivation of Products and Evolution of System-Family Assets

Within this work package we deal with several aspects of Application System Engineering. It deals with the questions how to create and validate products, built upon the platform. Creating new products is a main source of system-family evolution. However, the separate system artifacts at all levels of abstraction, evolve separately into improved versions. This is a second source of system evolution. In order to be able to deal effective and error-free with evolution we need to keep track of the connections

between requirements and artifacts. Therefore advanced requirements modeling and traceability techniques are needed.

Deriving a new member of the family, defined by a set of new requirements, means to:

- Identify those requirements that are common to all product variants and those which belong to specific variants.
- Select and adjust those parts of the system family architecture that can be reused in the new variant.
- To document and reuse the architectural decisions at the variation points which have to be taken to complete the variant architecture.
- Identify and possibly adjust the components along with an adequate configuration that can be used in the new product.

Capturing, documenting and maintaining the traceability information is labor and cost intensive. Traceability should thus be adjusted to product family specific needs and as far as possible automated.

Change to a product concerns modification related to defects or new requirements. It is necessary to analyze the affected assets and determine whether changes have impact on them. In general, we need to predict the properties of variants before actually building them. Thus we need to be able to determine impacts of change and how changes propagate.

Change may influence several system artifacts at different levels of abstractions such as components, architectural models, and requirements. Changing one artifact may require the adaptation of dependent artifacts, again at different abstraction levels. This dependency relationship refers to interactive change impact propagation. Change impact propagation is illustrated by references to potential changes required.

Product creation is based upon a selection of the specific functional or quality requirements. These selections determine the components that will form the application system, the parameters (properties, attributes) for instances of generic components and the interfaces that support the quality aspects. Based upon the selection the executable system has to be built. In [8] uses a process covering requirements capture, robustness analysis, design, implementation, testing and packaging the system.

Configuration may be dynamic at initialization and/or run-time. Any choice for the moment of configuration influences the possibilities of customizing, upgrading or adapting the components, architectures, and requirements.

Selection has to be performed efficiently and knowingly. In order to be able to do so, meta-information and version information about the components and parameters has to be available. Product versioning is often orthogonal to product variants. Versioning across different product variants has to be modeled and performed carefully.

Validation

All methods and techniques are validated on large case studies within the project. These case studies involve:

1. Communication Domain: Mobile phone families and Switch Maintenance families
2. Medical Domain, Image acquisition and management system families and Picture Information Management Systems (PACS)

3. Utilities systems
4. Air supervision systems
5. Car systems, Driver Information Systems and Automotive systems
6. Network servers for office equipment and web controlled embedded system architecture
7. Banking domain
8. Multimedia domain

As ESAPS cover a broad range of methods and techniques, each validation concerns only a small part of the complete effort. Each industrial partner defines for which aspects they want to measure improvement. Within ESAPS we provide templates to make experiment plans and document experiment results.

Summary and Conclusions

Many European industries recognize that a product family approach will be cost effective in future. ESAPS is a large European project, with the focus to the architecture and the development process of software for product families.

ESAPS is built upon earlier experiences within Europe and the USA. A collection of important topics is identified and these topics will be investigated within the project. In parallel validation is performed within a large range of industries to evaluate the work done within ESAPS. At the end of ESAPS we hope to have a collection of new, or refined techniques, approaches and methods that can be used in the development of software for product families.

Acknowledgements

The ESAPS project is ITEA project 99005 in the European Eureka Σ! 2023 Programme. The programme has a focus on information technology improvements.

The following ESAPS partners all made contributions to be able to give this overview:

The Netherlands:

- Eindhoven Embedded Systems Institute (EESI)
- Software Engineering Research Centre (SERC)

Finland:

- Nokia Research Center
- Helsinki University of Technology

Sweden:

- University of Karlskrona/Ronneby
- AXIS Communications AB
- Combitech Software AB

Germany

- Siemens-ZT
- Siemens-HS
- Robert Bosch GmbH
- Fraunhofer IESE

- Market Maker
- Universität Essen

France:

- Thomson-CSF
- Alcatel Corporate Research Center
- INRIA Rennes

Spain:

- Sainco
- European Software Institute (ESI)
- Unión Fenosa International Software Factory
- Universidad Politécnica de Madrid (UPM)

References

[1] ARES Web Sites, Available at Vienna <http://hpv17.infosys.tuwien.ac.at/ARES/> and Madrid <http://sirio.dit.upm.es/~ares/>
[2] PRAISE Web Site. Available at <http://www.esi.es/Projects/Reuse/PRAISE/>.
[3] Len Bass, Paul Clements, Rick Kazman, Software Architecture in Practice, 1998, Addison Wesley: ISBN 0-201-19930-0
[4] Paul Clements, Linda Northrop, A Framework for Software Product Line Practice, Version 1.0, September 1998, SEI Carnegie-Mellon
[5] E.W. Dijkstra, Notes on structured programming, O.J. Dahl, E.W. Dijkstra, C.A.R. Hoare, eds., Academic Press, London 1972.
[6] Philippe Kruchten, The 4+1 View Model of Architecture, IEEE Software pp. 42-50, November 1995.
[7] Marc H. Meyer, et al, The Power of Product Platforms: Building Value and Cost Leadership, 1997, Alvin Free Press: ISBN 0684825805
[8] Ivar Jacobson, et al; Software Reuse: Architecture Process and Organization for Business Success, 1997, ACM Press New York: ISBN 0-201-92476-5
[9] D. L. Parnas On the Design and Development of Program Families, Transactions on Software Engineering, SE-2:1-9, March 1976.
[10] Dilip Soni, Robert L. Nord, Christine Hofmeister, Software Architecture in Industrial Applications, Proceedings ICSE' 95, pp. 196-207, 1995.
[11] David M. Weiss and Chi Tau Robert Lai: Software Product-Line Engineering: A Family Based Software Development Process. Addison-Wesley 1999.

Product-Line Engineering

Paul Clements

Software engineering Institute, Carnegy Mellon University
Pittsburg, PA 15213, USA
clements@sei.cmu.edu

In the United States, product line activities can be divided between technology consumers and technology providers. Consumers are the practitioners, the people in companies who are turning out product families using common architectures, components, and other resources. Technology providers in the area of product lines produce either methods or tools, or sometimes both.

Technology providers representative of those providing methodological contributions include Lucent Technologies and the Software Engineering Institute.

Lucent is developing, using, and helping others to use a method called the FAST Process. FAST stands for Family-oriented Abstraction, Specification, and Translation, and it is a process for defining families and developing environments for generating family members. The steps of FAST involve finding abstractions common to the family, defining a process for producing family members, designing a language to specify family members, and then generating software for family members from specifications written in the family-specific language. FAST features an emphasis on families of systems, includes an explicit economic model for evaluating the economic feasibility of a particular family, explicit scoping to define what is in and out of the family, and rapid production of family members.

At Carnegie Mellon University's Software Engineering Institute, the Product Line Systems Program is one of the six technology programs currently under way. Its mission is to help organizations transition to a software product line approach and build a community of product line researchers and practitioners. Its work is based on the realization that technology alone is not enough to solve the problems, but that organizational approaches are required as well. Its primary work product is "A Framework for Product Line Practice," which defines a set of about 28 product-line-relevant practice areas in which product line organizations should be competent. The framework divides practice areas into three broad areas: those relating to software engineering, technical management, and organizational management. Each practice area is defined in terms of how it should be applied in the context of a product line (although most of the practice areas are in fact activities that are applied in generic form to any software development effort). Software engineering practice areas include Understanding Relevant Domains; Mining Existing Assets; Architecture Exploration and Definition; Architecture Evaluation; COTS Utilization; Software System Integration; Component Development; Testing; and Requirements, Elicitation, Analysis, and Management. Technical management practice areas are Data Collection, Metrics, and Tracking; Product Line Scoping; Configuration Management; Technical Risk Management; Process Modeling and Implementation; Planning and Tracking; Make/Buy/Mine/Outsource Analysis; and Tool Support. Finally, organizational management practice areas are Achieving the Right Organizational Structure; Operations; Training; Developing/Implementing Acquisition Strategy; Launching and Institutionalizing a Product Line; Building and

F. van der Linden (Ed.): IW-SAPF-3, LNCS 1951, pp. 253-254, 2000.

Communicating a Business Case; Funding; Market Analysis; Customer Interface Management; Technology Forecasting; and Organizational Risk Management. The SEI has also run a series 8 small focused product line practice workshops, written a series of product line industrial case studies, and this August will organize and host the first international software product line conference in Denver, Colorado.

Technology providers on the tool side are exemplified by the Microelectronics and Computer Technology Consortium (MCC). Here, product line architecture research is being carried out in the Software and Systems Engineering Productivity (SSEP) project, sponsored by shareholder companies including NCR, Raytheon, Motorola, Kodak, and TRW. MCC in general has a technology rather than process focus; in software engineering technology this tends to get embodied in tools and/or frameworks. The central theme of SSEP project has evolved to be something like "an exploration of the idea of architecture as organizing product lines." As a result, this has led to a structural rather then behavioral bias re architecture, with an emphasis on explicit linkage or correlation between architecture and other artifacts. The project now has a number of prototype tools that they are just now beginning to try (with participant companies) to use for real architectures. A key present activity is a joint case study with NCR to explore explicit representation of variability in architecture.

In addition to these large-scale initiatives, the current rich environment of start-up companies is providing some product-line-related technology, although little product line research by that name is being done. But many companies are doing it because they have to; they are simply not thinking of it as research. However, the entrepreneurial nature of small companies tends to make their results difficult to transition to the community at large, and difficult to scale up to meet industrial needs.

Author Index