

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

TOWARDS A TRUE SOFTWARE ENGINEERING DISCIPLINE

by

Daniel W. Drew, B.S.

THESIS

**Presented to the Faculty of
The University of Houston Clear Lake
in Partial Fulfillment
of the Requirements
for the Degree of**

MASTER OF SCIENCE

THE UNIVERSITY OF HOUSTON CLEAR LAKE

December, 1997

**Copyright 1997, Daniel W. Drew
All Rights Reserved**

UMI Number: 1387986

**Copyright 1997 by
Drew, Daniel Wayne**

All rights reserved.

**UMI Microform 1387986
Copyright 1998, by UMI Company. All rights reserved.**

**This microform edition is protected against unauthorized
copying under Title 17, United States Code.**


UMI
300 North Zeeb Road
Ann Arbor, MI 48103

TOWARDS A TRUE SOFTWARE ENGINEERING DISCIPLINE

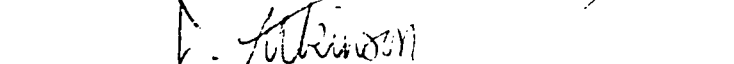
by


Daniel W. Drew


APPROVED BY


Sharon Andrews White, Ph. D., Chair


Charles McKay, Ed. D., Committee Member


Colin Atkinson, Ph. D., Committee Member


Robert N. Ferebee, Ph. D., Associate Dean


Charles McKay, Ed. D., Dean

DEDICATION

To my wife Karen and my son Galen. Thank you for your support and willingness to allow me the time for this achievement. May the obtainment of this goal bless you in return.

ACKNOWLEDGEMENTS

No accomplishment of any merit is ever achieved without the aid of others. The writing of this thesis is no different. This thesis is truly the end result of years of support and encouragement from many different individuals.

First I would acknowledge my Creator for the talents He has graciously endowed me that I am privileged to use. Next my father Dr. Dan Drew for his example and instruction as one of my undergraduate instructors and my mother Flowayne Drew for her guidance, instruction, and support. From my professional career David Weisman for encouraging me and mentoring me in the development of my technical writing skills. Finally to my instructors Dr. Sharon Andrews White, Dr. Colin Atkinson, Dr. Charles McKay, Dr. David Eichmann, and Mr. Kyle Rone for their willingness to teach.

ABSTRACT

TOWARDS A TRUE SOFTWARE ENGINEERING DISCIPLINE

Daniel W. Drew, M.S.

The University of Houston - Clear Lake, 1997

Thesis Chair: Sharon Andrews White, Ph.D.

The term “engineering” has been applied to software development for over twenty years yet there still is little progress in evolving the industry into a true engineering discipline. A fundamental reason for this lack of progress is that the paradigm of software engineering as defined does not completely capture the basic concept of engineering. This has led to software processes and methodologies that do not fully support engineering activities. To move toward a true software engineering discipline, there must be a basic paradigm shift in our understanding of software engineering accompanied with a related evolution of software processes and methodologies. Software engineering as it has been defined is reviewed and a new definition is presented that incorporates engineering principles currently omitted. How this paradigm shift can be implemented through the evolution of common processes and methodologies in use today is then illustrated.

Table of Contents

List of Tables	viii
List of Figures	ix
1.0 Introduction.....	1
1.1 Background	1
1.2 A New Paradigm for Software Engineering	4
2.0 Software Architectural Development.....	9
2.1 How Does this Differ from Standard Reuse Methods	11
2.2 Software Architecture	12
2.3 Summary	15
3.0 Making Software Development Process Models Engineering Based	16
3.1 History of the Evolution of Creational Software Process Models.....	16
3.2 Evolving to Engineering Based Process Models	22
3.3 Evolution of the Waterfall Model into an Engineering Based Process	24
3.3.1 System Feasibility.....	24
3.3.2 Software Plans, System Domain Boundary Identification, and Requirements Development.....	24
3.3.3 Select System Architecture.....	25
3.3.4 Adapt Standard Components and Unit Test	25
3.3.5 Integration	26
3.3.6 Implementation and System Test	27
3.4 Evolution of the Spiral Model into an Engineering Based Process	27
3.5 Evolution of Information Engineering into an Engineering Based Process.....	30
3.6 Summary of Software Development Process Evolution	33
4.0 Software Architectural Development and Patterns.....	35
4.1 Implementing Architectural Development Through Patterns	35
4.1.1 Architectural Patterns.....	36
4.1.2 Design Patterns	37
4.1.3 Idiom Patterns.....	38
4.2 General System Construction Based on Patterns.....	38
4.3 Pattern Format and Content	39
4.3.1 Context.....	41
4.3.2 Structure.....	41
4.3.3 Dynamics	42
4.3.4 Implementation	42
4.4 Required Additions to the Pattern Template	42
4.4.1 Companion Patterns	43
4.4.2 Preconditions	43

4.4.3 Constraints.....	43
4.4.4 Pattern Type	44
4.5 A Pattern Template that Supports Architectural Development.....	44
5.0 Making Software Development Methodologies Engineering Based	47
5.1 Evolving Fusion with Software Architectural Development	48
5.1.1 General Description of Fusion	48
5.1.1.1 Analysis.....	48
5.1.1.2 Design.....	49
5.1.1.3 Implementation.....	49
5.1.2 Fusion and the Software Development Process.....	50
5.1.3 Software Architectural Development Based Fusion	52
5.1.4 An Example of Engineering Based Fusion	53
5.1.5 Review of Engineering Based Fusion.....	65
5.2 Evolving Rapid Application Development with Software Architectural Development	66
5.2.1 Basic Need for Evolution to an Engineering Based Paradigm.....	67
5.2.3 Modifications to System Planning and Design Phase.....	67
5.2.4 Modification of the Construction and Cutover Phase	69
5.2.5 An Example of Engineering Based RAD.....	70
5.2.6 Review of Engineering Based RAD.....	72
5.3 Summary of Methodology Evolution.....	73
6.0 Conclusion.....	74
6.1 Existence of Scientific Basis to Support the Paradigm Shift	74
6.2 Related Work.....	76
6.3 Summary	76
6.4 Future Work	77
7.0 References	79
Appendix A: Requirements for Example Problem.....	83
Appendix B: Example Architectural Pattern.....	85
Appendix C: The RAD Life Cycle.....	90

List of Tables

Table 1: Major Process Models Mapped to Creational Meta Model	22
Table 2: Major Process Models Adapted for Architectural Development.....	23
Table 3: Comparison of Pattern Categorizations	37
Table 4: Pattern Description Template	40
Table 5: Pattern Template Supporting Software Architectural Development	44

List of Figures

Figure 1: Waterfall Model	17
Figure 2: Spiral Model	19
Figure 3: Information Engineering	20
Figure 4: Evolution of Process Models Addressing Specific Issues	22
Figure 5: Modified Spiral Model	28
Figure 6: General Software Construction Problem	38
Figure 7: Pattern Use in System Construction	39
Figure 8: Pattern Support for Software Architectural Development	46
Figure 9: Fusion Model Relationships	50
Figure 10: Process for Developing Software Products	51
Figure 11: Sale and Transaction Generic System Object Model	56
Figure 12: Standard Classes Adapted for Unique System	58
Figure 13: Scenario for Payment	60
Figure 14: Life Cycle for Payment as a Regular Expression	61
Figure 15: Data Model for Sale and Transaction Architecture	71
Figure 16: Process Decomposition Model for Payment	72

1.0 Introduction

1.1 Background

In 1975, Ross, Goodenough, and Irvine published a paper that captured the important underlying issues with respect to what at that time was the new concept of software engineering [RGI75]. Their description of software engineering was in terms of a process implemented upon a defined set of principles to achieve a specific set of goals. The motivation for their work was the realization that the notion of software engineering was ill defined except for the fact that software engineering implies the disciplined and skillful use of suitable software development tools and methods, as well as a sound understanding of certain basic principles.

The process described in Ross et. al., [RGI75] is composed of five steps. These steps are:

- **purpose** - define the requirements for a system
- **concept** - derive the architecture of a software system to satisfy the requirements
- **mechanism** - implement the software system (code/debug/tune)
- **notation** - define the means a user will employ to invoke the system capabilities
- **usage** - describe how the software system is controlled

The implementation of this process is governed by the application of seven guiding principles. These principles are:

- **modularity** - how to structure the software
- **abstraction** - identify essential properties common to different entities
- **hiding** - make inessential information inaccessible
- **localization** - methods for bringing related things together
- **uniformity** - ensure consistency

- completeness - ensure nothing is left out
- confirmability - information needed to verify correctness has been explicitly stated

This disciplined process founded on these principles is used to develop software systems that satisfy the following set of goals:

- modifiability - control can be exercised over subsequent changes to the system such that these changes do not degrade the system over time
- efficiency - the execution of system functionality occurs within required time constraints
- reliability - system implementation must prevent system failure as well as permit system recovery from unexpected failure
- understandability - the system must be understandable from the view point of all parties involved such as the user, developer, and manager.

A review of this landmark paper is important because these concepts of software engineering have had a major impact on how software engineering is perceived and defined.

Over the years there have been additions to the set of principles and goals as well as refinement to the process. However, the basic notion of software engineering being a disciplined process founded on a set of principles to achieve a set of goals has remained one of its underlying ideas. One cannot deny that this view of software engineering brings invaluable insight, but it has not led to a true software engineering discipline.

In the seventies, it was important to instill into the software industry that software development, particularly for large system development, cannot be successfully conducted without the exercise of a defined disciplined approach. This was a central theme in Ross et. al., [RGI75] and has continued to be the basis for work that strives to define and establish software engineering. Focus on this paradigm of software engineering has indeed brought the software industry to the general acceptance of the notion that the

development of quality software is strongly influenced, if not directly dependent, upon having a well defined process that is followed in a disciplined fashion. What this paradigm has failed to do is bring the industry to a full realization of software development as a true engineering discipline. To accomplish this will require a shift in our basic model of what software engineering is and an extension of the established foundation of disciplined software development processes with the adoption of new technologies and approaches to software development.

In Pfleeger, [Pfleeger91] the role of a software engineer is contrasted to the role of a computer scientist similar to the way a chemical engineer is contrasted to a chemist. The chemist is concerned with the investigation of chemical structures, interactions, and the theory behind their behavior. The chemical engineer is concerned with the application of these findings to solutions for a variety of problems. In short, chemistry as viewed by the chemist is the object of study where chemistry as viewed by the chemical engineer is a tool to be used to address a general problem.

In similar fashion, a software engineer is interested in using the findings of a computer scientist with regard to the structures, interactions, and theory of software as tools to solve a variety of problems requiring software systems as a solution. This analogy holds true regardless of the engineering discipline. There is always a scientific discipline focused at expanding the underlying theory and knowledge of the basic materials used by an engineering discipline as tools and resources to solve a specific problem.

The work of an engineer and scientist can be further contrasted in the following way. The scientist's primary goal is one of discovery. The engineer's primary goal is the specification and design of solutions to problems which are often repetitive in nature and the management of the construction of those solutions based upon the design. An engineer is concerned with the repetitive use of scientific discovery to create useful products as solutions to a recurring set of problems for a defined user community.

It is the emphasis on repetitive use of technology and recurring problem sets that is missing from the current software engineering paradigm. This omission has hindered the growth of the industry into a true engineering discipline.

Software development has remained fundamentally a process of creation and discovery. Granted, the progress in adopting software engineering practice has moved development activity away from ad hoc development towards more rigor and discipline but these disciplined processes only formalize the craftsmanship used to create new products. Little is found in these processes that encourage the use of scientific discovery as tools or the systematic reuse of properties or resources in the construction of software systems.

What is needed then to move software development towards a true engineering discipline is to first expand the current model of software engineering to emphasize the repetitive problem solving aspects of engineering and then evolve current processes and methods to this expanded paradigm of software engineering.

1.2 A New Paradigm for Software Engineering

A major theme that has emerged within the software community is that of repeatability or recurrence. Repeatability is the foundation of software process improvement as recognized in the levels of the capability maturity model [PCCW93]. Repeating designs are the basis for recognizing patterns in software development and documenting solutions to commonly occurring problems in software systems [BMRSS96]. Similarly, reuse of software artifacts is also often cited as the basis for improving software development productivity and quality [CN96].

These observations lead to the conclusion that the primary concern of software development is not in creative activity or new discovery but in routine development of solutions that are related and support repetitive use of processes and software artifacts. This observation is important because it substantiates the inclusion of routine development activity as part of the fundamental definition of software engineering.

The vast majority of engineering projects involve the routine design of solutions to recurring problems. Occasionally, the solution will require some degree of innovation to address a need unique to a specific instance of a common problem. Even rarer is the need to create or discover a new solution. This description bears little resemblance to the way software development is currently practiced in most organizations. Most current methodologies in use today to develop software can be described by the following model.

1. assimilate the application domain
2. abstract custom models for this domain
3. craft custom components [Best95]

The methodology may be very disciplined but it is still assumed that the primary activity is one of discovery, creation, and craftsmanship. If the development process is based on an engineering paradigm, then the methodology would be described by the following model.

1. Match the domain to standard architectural models
2. adapt the standard components associated with these models to meet domain requirements [Best95].

This second model emphasizes the use of existing experience and knowledge to derive solutions to common sets of problems. It suggests that software development is a routine activity that can build upon the solutions provided to previous problems. A reevaluation can now be made of existing software engineering definitions based on the notion that true engineering involves the routine development of solutions to a frequently recurring set of problems.

Pfleeger defines software engineering as a strategy for producing quality software [Pfleeger91]. Sommerville describes software engineering as being concerned with theories, methods, and tools to produce software products in a cost-effective way [Sommerville96]. We can combine these two descriptions of software engineering with

the description in Ross et. al, [RGI75] to develop a comprehensive definition of the current perception:

“Software engineering is the disciplined and skillful use of software development tools and methods to implement a strategy characterized by the notions of modularity, abstraction, localization, hiding, uniformity, completeness, confirmability, to produce quality products which are understandable, reliable, efficient, and modifiable, in a cost-effective manner.”

Since the goals are qualifiers for quality and the principles characterize the strategy, we can shorten the definition to read as follows:

“Software engineering is the disciplined and skillful use of software development tools and methods to implement a strategy to produce quality software products in a cost-effective manner.”

It is interesting to note the ease with which these three descriptions can be blended considering that they span twenty years. This is more evidence of how closely the software community has kept with the original notions of software engineering. These statements as definitions are more statements of desired results than definitions of engineering activity. We have already shown that the primary activity of engineering is to provide solutions to recurring problems. Furthermore, we have established that the discovery of software properties and underlying theories lie in the realm of computer science. As such, a discussion as to whether such a set of properties exists is outside the scope of this thesis. What is of importance is that the capture of this knowledge is accomplished in ways that support its repeated use in multiple products.

Our new definition of software engineering is as follows:

“Software engineering is the disciplined and skillful use of software development tools and methods applied to recurring problems to produce useful products in a cost-effective manner.”

This definition is designed to emphasize two important points that transform the initial definition into one closer to true engineering. First, the essence of engineering is routine development of products for repeating problem domains. Second, the resultant product is useful to the user.

Engineering by its nature is routine. This does not imply that engineering projects are always simple or straightforward. Routine implies that there is a marked lack of invention. Rather, the project is one of applying proven practice to a specific problem. The lack of this one property in software development has contributed more to inhibiting the growth of software development into a true engineering discipline than any other issue. The processes and methods in use today are still based on a creational paradigm.

The second area emphasized in this definition is that the products produced by an engineering activity must be useful. A current trend in all industries is to focus on the notion of quality for products. While this is a very important concept and the adoption of total quality management principles has had a major positive impact on industry at large, it is often difficult to fully understand what quality means with respect to a software product. There is also a tendency to confuse quality with notions of products that are best of breed, provide the most options, or are superior to all other products. These are limited notions of quality.

Being focused on usefulness still embodies the concept of quality while maintaining the proper perspective of the goal of engineering. A product that is engineered can usually be characterized as one that is a practical solution that fits the needs of the user. For software, the definition of “user” can become complicated. Users of a software product are the individual who uses the system in support of work activity, the maintainer of the software system, and management responsible for overseeing the system’s use and upkeep. With this in mind, one can see how quality and usefulness can be synonyms. If we use the four goals from Ross et. al. [RGI75] as a description of quality, we can see how these same goals describe the usefulness of a system. Understandability and modifiability

support the system maintainer while reliability and efficiency support the end user. All four goals indirectly support management responsibility.

Our paradigm for software engineering must then be more than a principled disciplined process directed toward a set of goals. The paradigm must be one that supports the application of an established base of knowledge for the routine development of useful software products. We investigate these ideas further in this thesis.

Chapter 2 introduces and describes software architectural development as the basis for evolving software processes and methodologies to an engineering based paradigm.

Chapter 3 presents a brief history of the evolution of software process models. We then show how process models can be transformed into models based on an engineering paradigm. Chapter 4 describes a general approach for the implementation of software architectural development using software patterns. Chapter 5 describes how the general approach in chapter 4 is applied to the transformation of methodologies to an engineering based paradigm. Chapter 6 revisits the notion of software engineering with respect to the existence of scientific support and summarizes the findings of this research.

2.0 Software Architectural Development

Having established a definition for software engineering and a paradigm that reflects a truer nature for an engineering discipline, the next question to ask is what changes are needed in the supporting software technology? Technology is used here in its most general definition to include tools, processes, and methods that are used in the development of software systems. The key to understanding the needed changes in technology is to understand the differences between the two development models presented by Best [Best95].

The first model is one that covers what is called process-based, non-architectural approaches. This model is referred to as process-based since it is a meta model for existing software development processes such as the waterfall model. It is considered non-architectural due to the fact that the model does not require the use of architecture to guide development activity. It is, however, quite probable that an architecture, formal or informal, will be created at some point in the software development process. We will refer to this model as the creational model to emphasize the fact that the primary work activity is the creation of new software artifacts. This model describes software development in three phases.

1. **Assimilate the application domain.**

This phase would establish a thorough comprehension of the domain for which a system would be built. It implies that there is incomplete knowledge about the domain and models or artifacts that might exist must be enhanced before one could begin to match domain needs to a developing system.

2. **Abstract custom models for this domain.**

This phase implies that any system framework or design must be created specifically for the analyzed domain.

3. Craft custom components.

Here again, all components¹ to be used in the software system must be created specifically for this project. Although there may be an attempt to reuse components from other systems, the methods this model describes provides little to no support for this activity. Additionally, current approaches to reuse have had little real world success. Current approaches to reuse have been compared to archaeological digs which only occasionally reveal hidden treasures.

Methods that can be described by this model provide little or no support for engineering activity. The emphasis is one of discovery and creation. Routine development is hindered because there is no incentive to build upon past experiences or use artifacts from similar systems.

The second model presented for software development methods is called architectural development. This model is based on the assumption that software architectures can be captured in a manner that allows them to be recognized as a potential solution to a stated problem in a given domain. It is this model of development methods that provides the basis for engineering software systems. This model describes the development process in two phases.

1. Match the domain to standard architectural models.

Generic architectural models would exist that would allow analyst to quickly match specific domains and domain terminology to a family of applications.

2. Adapt the standard components associated with these models to meet domain requirements.

¹ Components in this thesis refer to any item used in software development, i.e. design models, code, test cases, etc.

Associated with the architectural models would be components used to implement the model. These components would be adapted to support the unique specifications of the domain.

The result would be a customized solution based on a standard architecture derived from a standard library of components. The ability to capture standard architectural models supports the notion of routine development. It eliminates the creational approach and replaces it with one that relies on handbooks of standard practice. We will adopt software architectural development as the name for this approach for software development.

2.1 How Does this Differ from Standard Reuse Methods

The difference between architectural development and reuse as practiced today is that architectural development bases the selection of components on established system architectures that map to domains. Reuse as practiced today considers each component individually and attempts to discover elements within the component that support the creation of a new system. The components do not support an established architecture. The process of identifying reuse components is not tied to architectural models as well. Identification relies upon component descriptions and semantic or structural bindings with the developing system.

Software architectural development on the other hand provides support for the systematic reuse of software components. Issues of component capability, interface, and control are already addressed through the architectural model for the system.

A major difference between standard reuse and architectural development is the lack of solid methodology and process support for standard reuse. Adoption of an architectural development based methodology makes reuse of components part of the standard method of system development.

2.2 Software Architecture

The proper application of software architectural development requires an understanding of what software architecture is. The notion of architecture in software is relatively new and not yet fully matured. Therefore, it is necessary to explain what we mean by software architecture and how the concept applies to the application of software architectural development.

Shaw and Garlan state that software architecture involves the description of elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on these patterns [SG96]. The focus of software architecture is on reasoning about the structural issues of a system [CN96]. These structural issues include:

- composition of components
- global control structures
- protocols for communication
- composition of design elements
- physical distribution
- scaling and performance
- dimensions of evolution
- selection among design alternatives [SG96].

A software architecture then is a defined software structure that captures high level design definitions in sufficient detail that supports analysis of its usefulness, evolution into hybrid designs, and can serve as the basis for system development.

The natural trend in the combined work on software architecture seems to be moving towards a paradigm of software development based on principles of architecture [CN96]. This observed trend is the basis of this thesis. The goal of identifying a software architecture is to capture the structural commonality among members of a program family so that high-level decisions found in each member of the family need not be re-invented,

re-validated, and re-described. This aspect of architecture is the foundation of software architectural development. The nature of architecture serves as the link between a problem statement in a given domain and the selection of a solution proven to be viable through use in previous similar applications.

Software architecture can be described from at least three perspectives, the functional partitioning of its domain of interest, its structure, and the allocation of domain function to that structure [KBAW94]. The description of system functionality links the architecture to specific domain applications, the description of structure enables the architecture to be used as the basis for system development, and the allocation of domain function to structure serves as the bridge between the domain based problem statement and the programming based solution.

Having system structures described in a manner that captures the commonalties within a program family and having a clear link between that structure to stated domain problems support the creation of processes and methods that address the repetitive nature of software development which is an essential aspect of establishing a software engineering discipline. It is this aspect of architecture that is expanded on in this thesis. We hope to show how basing software development on the use of architectures will allow us to evolve the current traditional creation based methods to methods that directly address the repetitive nature of software development thereby moving us towards a more correct concept of software engineering.

At present there is little consensus as to what detail should be included in an architecture, how it should be represented, and the terms used to define and describe architecture [CN96]. Research addressing these issues with architecture can be roughly divided into four categories:

- architectural description languages - the development of languages which provide better ways to document and communicate architectures

- **codification of architectural expertise - the cataloging and rationalization for the variety of architectural principles and patterns developed through software practice**
- **frameworks for specific domains - the development of architectural frameworks for a specific class of software.**
- **formal underpinnings for architecture - the development of formalisms for reasoning about architectures [SG96].**

The actual detail needed for a full definition of architecture is a critical concern but not part of the main focus of this thesis. Our concern is for the establishment of a technique that can use an architectural definition as the basis for software development. We will show an approach that can accommodate any form of architectural representation and assume that the proper level of detail is available in the architectural definition.

At the very least however, we would expect a software architecture to provide:

- **identification of all major system components**
- **detailed information about the functions provided by components and the context of their use**
- **detailed information about the structure and dynamics of each component**
- **identification of all connections and relationships of the major components**
- **detailed information about component connections that fully describe the structure and dynamics component interconnections**

As one can see, the full treatment of software architecture is a major work in itself and well beyond the scope of this thesis. Our focus is on the use of software architecture in the evolution of processes and methods in common use today towards what we feel to be a more complete definition of software engineering. As such we will not elaborate on the various issues of architectural detail and representation. Examples of the use of

architecture within this thesis will be simple and usually only partial representations of architecture. The intent is to demonstrate a technique that can be used to incorporate the use of software architectures within existing processes and methods to adapt them for better support of repetitive engineering activity.

2.3 Summary

Two meta process models have been presented that define the family of software development process models in current use today and a proposed family of software development process models. Current software development models fall in the family of creational software process models. Since they are creational based processes, they do not adequately provide the support necessary for evolution towards software engineering.

The second defined meta model is the software architectural development model. This family of software development models promotes the establishment of defined system architectures which serve as a basis for software development.

Chapter 3 will address software architectural development models in further detail. A brief history of the evolution of modern day software development processes is presented to show how advances in software development processes are not moving towards software engineering. Then representative process models will be mapped first to the creational meta model. Next we will show how these models may be modified to provide support for software architectural development thereby adopting what we believe to be a more accurate definition of software engineering.

3.0 Making Software Development Process Models Engineering Based

3.1 History of the Evolution of Creational Software Process Models

Software process models determine the order of the stages involved in software development and establish the transition criteria for progressing from one stage to the next [Boehm88]. There has been a continual evolution of process models which began as early as the late 50's. The establishment of process models as well as their evolution has been driven by the increasing complexity of software systems. In fact, one could conclude that the root cause of the "software crisis" is our inability to manage the impacts on software development due to the complexity of software systems.

Process models address two simple questions. These questions are:

1. What shall we do next?
2. How long shall we continue to do it [Boehm88]?

Although simply stated, answering these two questions is proving to be the key to achieving control over software development and the continually increasing software complexity issue [Humphrey89].

The earliest of process models was the code-and-fix model. This model has two steps:

1. Write some code.
2. Fix the problems in the code.

It quickly became self evident that relying upon this model of software development allowed systems to become unstructured and expensive to maintain. This led to the adoption of the stagewise model [Benington56]. This model defined a set of sequential stages for software development. These stages are:

1. operational plan
2. operational specifications
3. coding specifications
4. coding
5. parameter testing
6. assembly testing
7. shakedown
8. system evaluation

The waterfall model, figure 1, developed in 1970 was a refinement of the stagewise model [Royce70]. It provided two primary enhancements to the stagewise model:

1. Recognition of feedback loops between stages with guidelines to minimize rework involved in feedback across many stages.
2. The incorporation of prototyping in the software life cycle that ran parallel with requirements analysis and design.

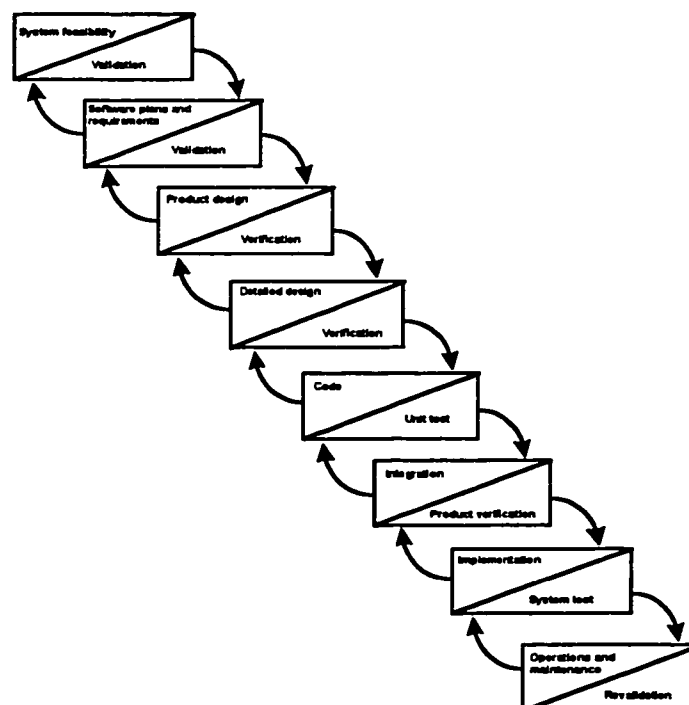


Figure 1: Waterfall Model

As systems became larger and more complex, it became impossible to fully define system requirements and follow a straight sequential process model. The prevailing requirement for system flexibility and the inability to fully anticipate all system requirements up front led initially to variations on the waterfall model by introducing the notion of incremental development. The waterfall model still served as the basic process model but the system is developed in smaller increments and integrated over time.

The evolutionary development model is another process model developed in response to the perceived rigidity of the waterfall model [MJ82]. This model allows the system to evolve over time and has relied heavily upon fourth-generation languages for its implementation. This model supplies rapid initial operational capability and provides a realistic operational basis for determining subsequent product improvements [Boehm88]. There is a significant risk with this model however of development slipping back into the old code-and-fix model.

With advances in automated code generation technology came the transform model of software development [BCG83]. This model relies on formal specification of the system that can be transformed into the software product.

The spiral model, figure 2, is another refinement of the waterfall model [Boehm88]. The primary focus of this model is the management of the risk associated with software development. This model cycles through the four primary activities of determining objectives, evaluating alternatives, development, and planning. Risk assessment is incorporated in the evaluation activity for each iteration of software development activity.

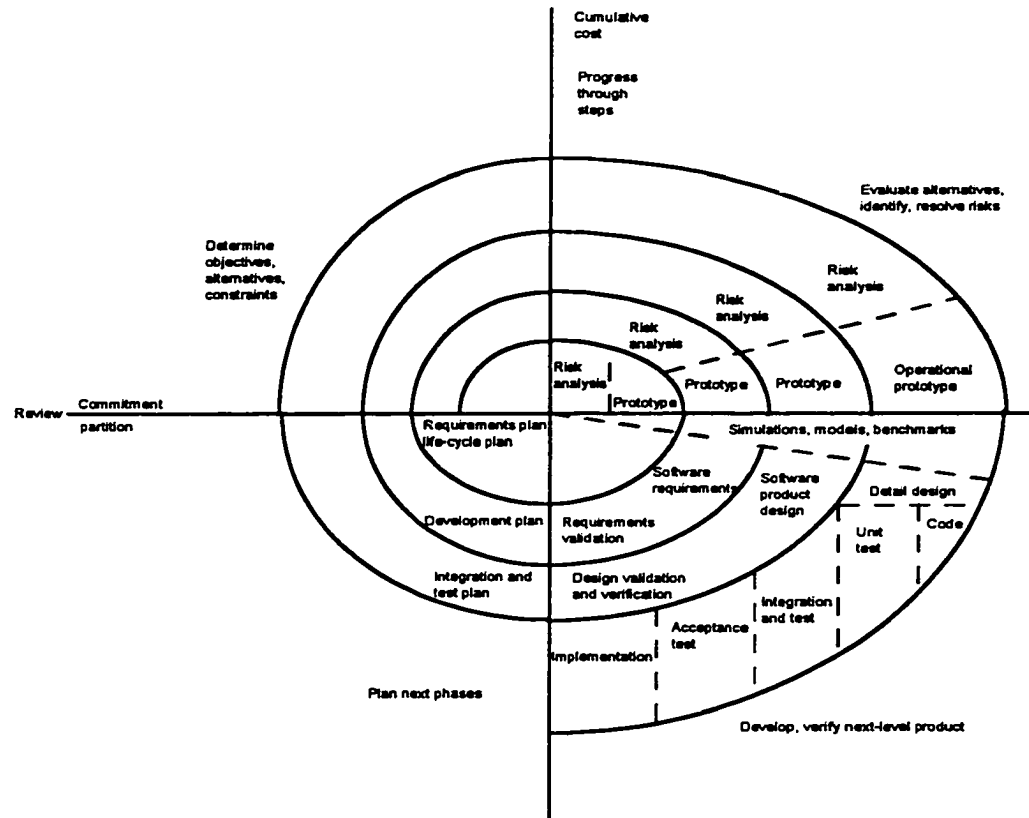


Figure 2: Spiral Model

Up to this point we have shown the evolution of general process models. One additional model should be noted that was developed specifically for business system development. This model was developed by James Martin in a time frame roughly parallel to the spiral model. The model is called Information Engineering, figure 3 [Martin90]. This model is based on the realization that individual systems within a single enterprise are often highly integrated with respect to the enterprise business processes. Information Engineering expands development activity from single system based development to an entire enterprise in a top down fashion to help assure proper integration of individual business systems. Information Engineering will be described more fully in section 3.5.

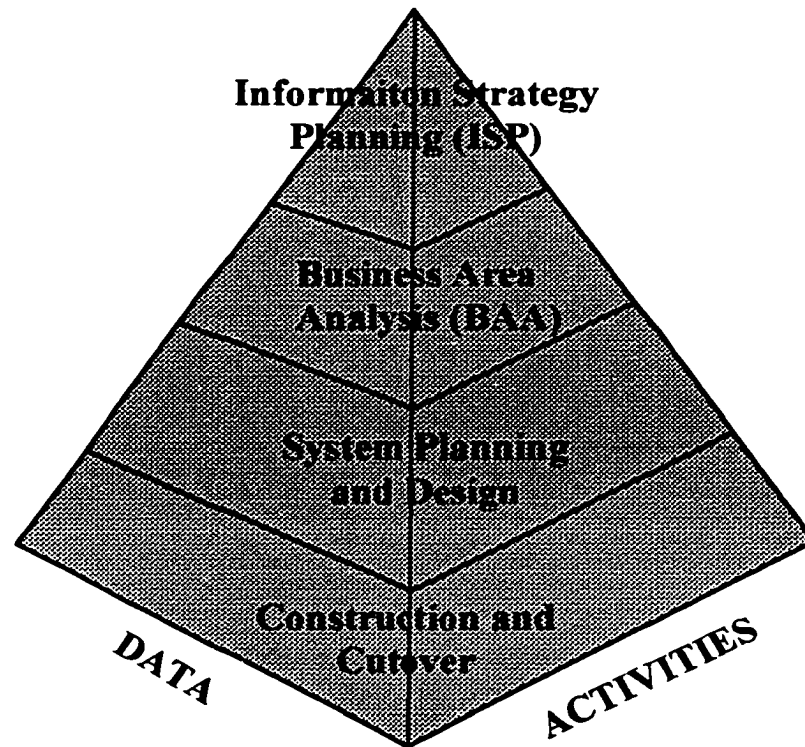


Figure 3: Information Engineering

Figure 4 places the evolution of software process models in perspective. We can see that each evolutionary step was intended to address a specific software development issue driven by ever increasing system complexity. The basic paradigm of creational development has remained constant as each process model evolved from its predecessor. Each model endeavors to create a system specification, system design, and then an implemented system. The basic premise for the spiral model, for example, is that several iterations of analysis and design supported by throw away prototyping must occur before there is enough comprehension of the requirements to actually build the system. We can also see that although this evolution has brought considerable ability to control system development, the price has been one of efficiency. This is evidenced in the emergence of the divergent path of evolutionary development. It is not until the control of system complexity brought focus to domain identification that significant attention was given to the notion of providing for routine treatment of repetitive systems in process models. We

see this shift best demonstrated in the emergence of Information Engineering and the concept of Domain Engineering.

Information engineering shows the significant role domain and domain understanding plays in software development. The top two activities shown in figure 3, which are the ISP and BAA, have as their purpose the identification of the overall domain of the enterprise as well as specific business domains the comprise the enterprise. What is not present in Information Engineering is acknowledgment of the fact that most domains can be described with generic models [PS94]. This suggests that families of software architectures could be constructed to address domains and that these architectures could be used as the basis for software system development with the bulk of the development activity devoted to adapting general components to specific needs of a unique instance of a domain. The existence of families of systems and the ability to develop specific solutions from general components is the essence of engineering and a fundamental theme in this thesis.

The primary intent of domain engineering is the development , capture, and evolution of knowledge and assets for a family of systems [WE96]. Assets that are developed meet common requirements across the domain and are tailorable for specific application needs. We can clearly see that in domain engineering we are adopting a major shift in the fundamental paradigm for system development that takes us away from creation based development to one based on the reuse of a family of assets. Software architectural development is closely related to domain engineering. Domain engineering could serve as the technique used for the initial identification of architectures and software architectural development used as the back end technique for system development.

We conclude from this analysis of the evolution of process models that the evolutionary development of software processes is not leading toward an engineering discipline. The move toward engineering will require a revolutionary shift driven by domain engineering and software architectural development.

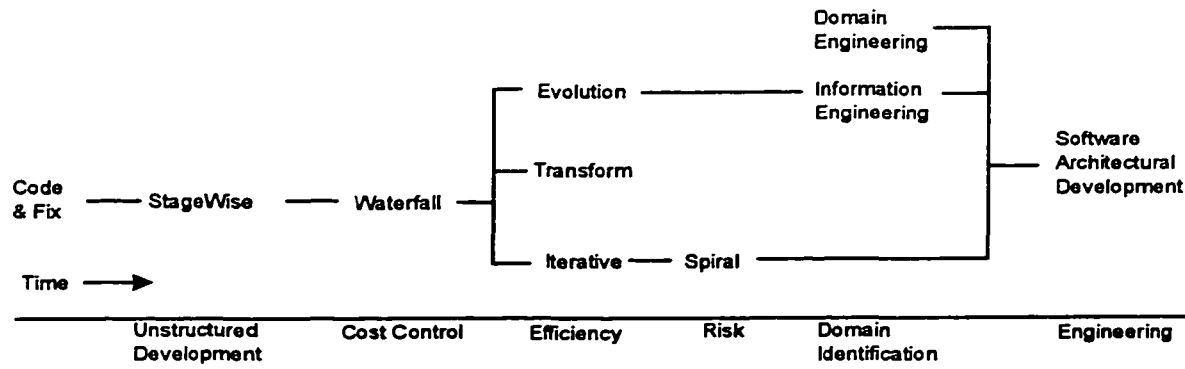


Figure 4: Evolution of Process Models Addressing Specific Issues

3.2 Evolving to Engineering Based Process Models

Table 1 maps the three major process models in use today to the general creational process. As has been shown, each basic process structure was created in response to specific problems encountered in the development of large complex systems and it is desirable to maintain those features of the process model. They are not, however, evolving towards an engineering paradigm based on the fundamental concept of routine system development.

Creational Meta Model	Waterfall Model	Spiral Model	Information Engineering
Assimilate Domain	<ul style="list-style-type: none"> • System feasibility • SW plans and requirements 	<ul style="list-style-type: none"> • Risk analysis • Concept of operation • Simulations • SW requirements • Requirements validation • Operational prototype 	<ul style="list-style-type: none"> • Information Strategy Planning (ISP) • Business Area Analysis (BAA)
Create Custom Models	<ul style="list-style-type: none"> • Product design • Detail design 	<ul style="list-style-type: none"> • SW product design • Design validation & verification • Detail design 	<ul style="list-style-type: none"> • System Planning & Design
Create Custom Components	<ul style="list-style-type: none"> • Code • Integration • Implementation 	<ul style="list-style-type: none"> • Code • Unit test • Integration & test • Acceptance tests • Implementation 	<ul style="list-style-type: none"> • Construction & Cutover

Table 1: Major Process Models Mapped to Creational Meta Model

What is required to move these models toward an engineering based paradigm is the removal of segments of the process model that rely on discovery and creation and replace it with process activity that supports routine development based on defined system architectures and standard components

Table 2 summarizes the changes that must be made to the stages in each respective process model in order to support architectural development. Stages that involve the creation of design or code are replaced by stages that identify and select standard architectures and modify the underlying standard components that are used to implement the architecture.

Requirements must still be established for the system. However, this process now becomes one of domain analysis with the intent of bounding the system to be developed. This analysis flows into architecture identification which takes the place of traditional preliminary or high level design. The evolution of each process model into an engineering based process is fully described in the following sections.

Architectural Development Meta Model	Waterfall Model	Spiral Model	Information Engineering
Match domain to standard architectural models	<ul style="list-style-type: none"> • System feasibility • Bound system within the domain and develop requirements • Select appropriate architecture for system 	<ul style="list-style-type: none"> • Risk analysis • Bound system within the domain and develop requirements • Requirements validation • Select appropriate architecture for system 	<ul style="list-style-type: none"> • Information Strategy Planning (ISP) • Business Area Analysis (BAA) • System Planning & Architecture selection
Adapt standard components	<ul style="list-style-type: none"> • Adapt components used to implement the architecture • Integration • Implementation 	<ul style="list-style-type: none"> • Adapt components used to implement the architecture • Integration & test • Acceptance tests • Implementation 	<ul style="list-style-type: none"> • Adapt components used to implement the architecture • Cutover

Table 2: Major Process Models Adapted for Architectural Development

3.3 Evolution of the Waterfall Model into an Engineering Based Process

The waterfall model provides a sequential process for the development of a software system, figure 1. The strength of this approach is the provision of intermediate checks of interim products as the system is being developed. These checks allow for greater control over system development by providing insight into the steps of the process. Greater product quality is also supported at lower cost because the interim checks drive out errors introduced earlier in the development life cycle.

The goal of evolution to an engineering based process model is to retain the basic characteristics of the model while replacing the creational aspects of the model with steps based on architectural development. The following describes how each step in the process model would be effected by an evolution to an engineering based paradigm.

3.3.1 System Feasibility

This step remains essentially unchanged. There will always be a need to assess the feasibility of any request for software development. The actual criteria used to determine feasibility will vary based on the domain of interest and the functional need required by the requester.

3.3.2 Software Plans, System Domain Boundary Identification, and Requirements Development

There is a significant shift in the activity associated with establishing system requirements. Since we are dealing with routine development, it is assumed that there exists an adequate description of the domain. The task of requirements definition then is to capture the unique characteristics of this particular request that will direct the selection of a specific architecture and set of components. These requirements would be similar to requirements that guide the selection of a specific architecture for a bridge. The engineer will be concerned with defining the anticipated stress and load requirements and similar attributes that will guide the selection of bridge structure and materials to be used. In similar

fashion, a software engineer will be interested in capturing requirements related to response time, number of users, and data volume. The functionality of the system is already known since this is a routine development problem.

The feed back loop to system feasibility still exists. The information gathered in this step will further refine the feasibility study by clearly identifying the system boundaries within the domain and establishing limits on system functionality thereby validating that the requirements address the issues identified in the feasibility study.

3.3.3 Select System Architecture

This step replaces Product Design and a large portion of Detailed Design. Since the development is routine engineering activity it is assumed that a set of proven architectures are available as potential solutions. This step will result in the selection of a specific system architecture and an initial set of standard components that best support the unique specifications and general functionality.

Custom designs do not need to be created. The engineer can rely upon approaches taken by other engineers that can be analyzed for their appropriateness based on principles supplied by computer science. Traditional design activity will be needed to design any modifications and additions needed to the standard set of components that are necessary to support unique system specifications.

A feed back loop still exists to requirements definition. The chosen architecture and the required modifications to supporting components are still verified against the stated user need identified in the previous step.

3.3.4 Adapt Standard Components and Unit Test

This step replaces Code and Unit Test. The existence of standard components would eliminate the need for custom development of routine components. However, there still may be need for additions, new components, which would be created for a given system.

Hopefully, the number of additions will be small with the goal being to achieve zero new components. Since the components support an established architecture, interface code or glue would also exist. This means that the only coding required would be adjustments required for unique system requirements. The need for special code may be further reduced by creating configurable components. This would be accomplished by incorporating different features within a component that have been proven desirable for systems of a specific type.

Adoption of this approach to development does not eliminate the need for unit testing. However, there is a clear advantage in that a library of trusted standard components is being used as the basis for development. This provides an opportunity to minimize the risk of component failure and support the establishment of standard unit tests thereby reducing the often overlook risk of the faulty test procedures or test products. Unit test will become more focus on areas of modification.

There is an obvious and very large assumption associated with this step. It is assumed that a library of standard, reusable, and configurable components exists that implement the domain architecture and support the routine development activity.

3.3.5 Integration

The task of system integration is simplified under this scheme. The architecture and standard components provide an integrated structure that is already proven. Integration can follow a known progression for piecing the system together. This aids in system planning and testing as well as actual component integration. Testing can focus on those areas where unique code changes were required and rely on trusted regression testing for general system testing.

Once again, it is assumed that standard architectures and standard components exist to support system construction. The fewer components that are available for reuse and an increase in custom modification will increase the effort required in this step.

3.3.6 Implementation and System Test

This step remains essentially unchanged by the adoption of an engineering paradigm. The system is still delivered and deployed like systems constructed under a creational paradigm. There should be, however, less risk in delivering an unreliable system to the user since a routine solution has been engineered. Not only is the system based on trusted software components, our set of standard components should also include established test plans, procedures, and test sets that have also been proven over time through routine use. This allows new tests to focus on modifications required to support unique system requirements.

3.4 Evolution of the Spiral Model into an Engineering Based Process

The spiral model is a risk driven approach to software development that can accommodate most other models of software development as special cases [Boehm88]. This model is best applied when there exist multiple solutions or when significant risk to project success exists. Once again, the goal of evolving this model into an engineering based model can be achieved by replacing the creational aspects of the model with architectural development based activity while leaving the general characteristics of the model intact. As shown in figure 2, it is the bottom right quadrant that will be most effected as the model evolves to an engineering paradigm. It is also reasonable to assume that there will be fewer iterations in development due to the use of established architectures and component sets.

The modifications recommended to the spiral model, like the recommendations for the waterfall model, are intended to adapt the model for routine system development. As such, the same set of assumptions stated for the waterfall model apply to this model. Specifically, there exists adequate domain definitions, a library of standard architectures, and a set of standard, reusable, and configurable components and associated testing documents and test data sets.

Figure 5 depicts the changes that must occur for the spiral model to become engineering based. The risk driven aspects of the model remain the same. However, since our basis for development are sets of well understood architectures and related components, some traditional sources of risk are minimized or eliminated.

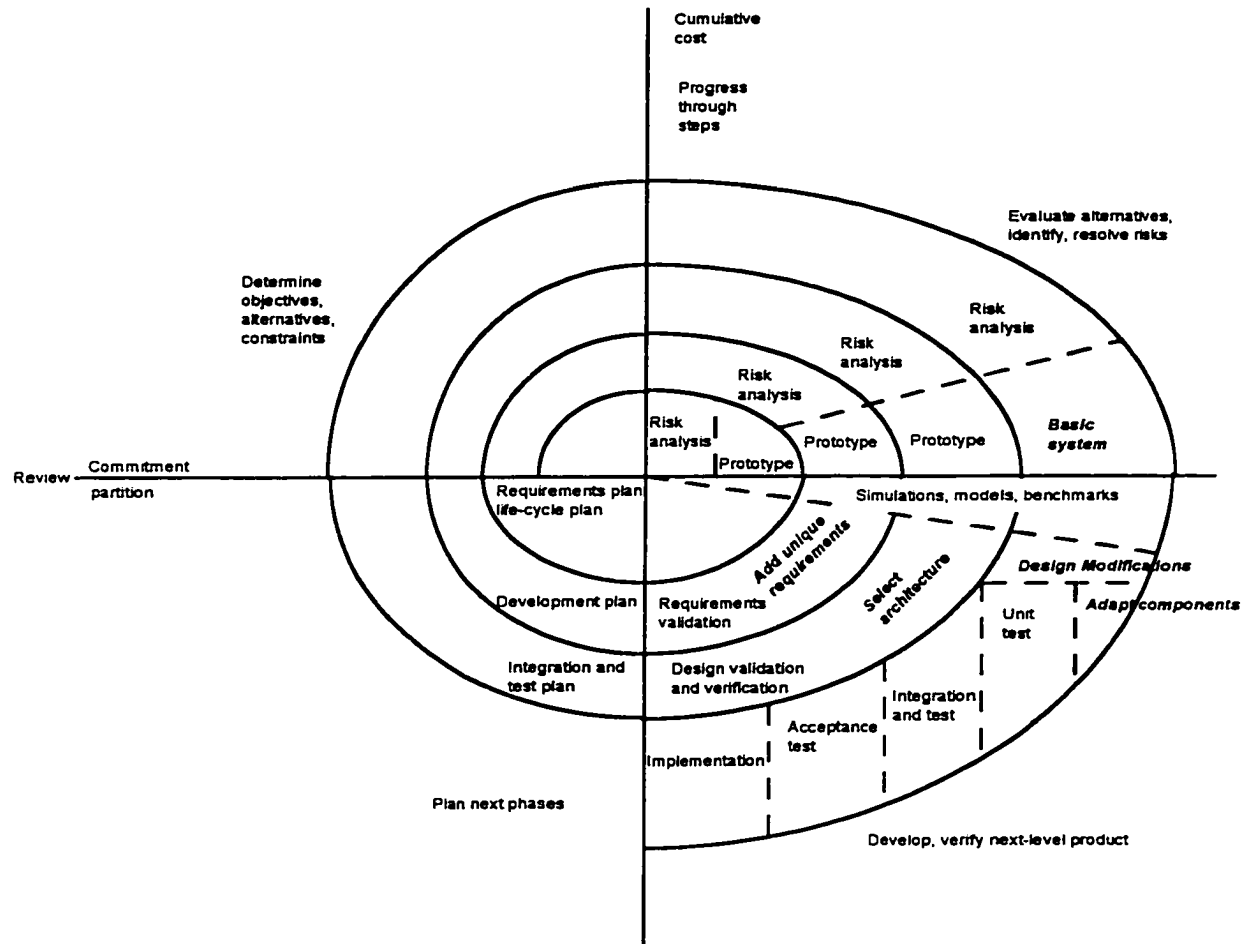


Figure 5: Modified Spiral Model

Risk analysis varies in each spiral. The type of risk being assessed depends upon where development is within the execution of the process. Initial risk analysis is concerned with system feasibility and the capture of true requirements. Risks associated with software development activity become more central to the analysis activity as the process moves into later stages. Infusing architectural development based activity into this model enhances our ability to manage risk within a software development project. For risk

analysis supporting front end activity, architectural development brings better definition of domains and structures in the form of architectures that clarify issues of system feasibility and requirements definition. The ability to judge between routine understood functionality and unique variants becomes easier thus making the areas of risk more manageable. Risk analysis in the latter stages of development dealing with the actual product construction are supported by providing proven approaches to solution construction and past experience to support project costing and scheduling. Component functionality is also known going into the project so the risk of developing software that does not satisfy the requirements is reduced.

The original intent of the first spiral in the model is to define the basic operation concepts of a new system. It is implied that this operational concept model does not exist and must be created. An engineering based model would replace this first spiral with one that drew from an existing domain model to scope and bound the specific area of the domain that will be the target of a new system. This will allow the selection of a set of architectures that further define the operational concepts and that will also serve as a basis for prototypes. These prototypes will be used to evaluate each architecture and select the best fit for implementation.

The second spiral then becomes one of identifying the unique requirements that will be included into the general system functionality provided by the base architecture. These requirements are added with the scoped domain model to support the final selection of an architecture and standard component set to use as a basis for development.

The third spiral is the actual selection of the base architecture and standard component set for system development. Validation and verification is performed against the selected architecture to assure that the requirements are satisfied.

The final spiral is similar to the steps in the waterfall model described in the previous section. There would be no operational prototype since the selected architecture with its set of components provide a functional base system as a platform for system development.

3.5 Evolution of Information Engineering into an Engineering Based Process

James Martin describes the difference between software engineering and information engineering in the following way: “Software engineering applies structured techniques to one project. Information engineering applies structured techniques to the enterprise as a whole or to a large sector of the enterprise. The techniques of information engineering encompass those of software engineering in a modified form” [Martin90]. The definition of information engineering then is as follows: “the application of an interlocking set of formal techniques for the planning, analysis, design, and construction of information system on an enterprise-wide basis or across a major sector of the enterprise” [Martin91].

The emphasis in information engineering is the same as in the traditional view of software engineering which is the establishment of formal techniques for the creation of new software systems. The major distinction between this process model and the two previous models is that this model acknowledges that systems are interconnected and provides a formal means of identifying those connections and developing systems within the defined framework.

To summarize, information engineering is a process designed to control the enormous amount of information that drives software development within an enterprise in a manner that permits the development of individual software systems that are appropriately interrelated. It is this fundamental concept that will be preserved as the model is evolved to support the engineering paradigm.

The process steps for information engineering are depicted using a pyramid, figure 3. The pyramid symbolizes the explosive increase of information that must be managed as the detail of actual system implementation increases as development activity moves from strategic planning for the enterprise to actual construction of the many systems that support the enterprise.

The information engineering process model is comprised of six steps [Martin91]. These steps are:

1. Information Strategy Planning (ISP)

This step creates plans for information systems that are in line with the strategic business plans for the organization. The strategic plan produced in this step relates future technology to how it could affect the business, its products or services, or its goals and critical success factors. This plan is used to guide and prioritize expenditures on computing to maximize the effectiveness of information systems' contribution to corporate objectives.

2. Business Area Analysis (BAA)

The enterprise is subdivided into business areas. Business areas are identified in information strategy planning by creating an overview model of the enterprise. Business areas usually are derived from identification of major enterprise activity defined by a business process model.

The objective of business area analysis is to understand what processes and data are necessary to make the enterprise work and to determine how these processes and data interrelate.

3. Individual System Planning

The goal of the first two steps is to model the business activity of the enterprise and divide the enterprise in an appropriate fashion in order to facilitate system development. This step turns attention to the technology that will be used to implement the business model. Each business area may have one or more underlying information systems. This step plans the development of these systems.

4. System Design

Each system is designed and developed as an independent project. Interconnection models are created and maintained to assure that systems are compatible and support the interrelated business activity between business areas. This step designs the data model for a specific system and the software components for that system.

5. Construction

This step is the coding and testing of a specific system.

6. Cutover

This step is the delivery of a new system and its deployment within the enterprise.

All models and artifacts developed in the process are maintained in a common repository. The process emphasizes the need for relating all developing models and reusing common aspects between developing systems. Automated support for information engineering is also considered critical for success due to the large volume of information about the enterprise that must be managed.

We can consider evolving this software development process model into an engineering paradigm at two levels. We can modify the system planning, design, and construction step thereby evolving system development and we can also modify the ISP and BAA activity thereby evolving the enterprise modeling activity.

The system development level of this model is evolved by adopting architectural development as the basis for system design and construction. Unlike the waterfall and spiral model, this model is tied very closely to a specific methodology for its implementation. Due to this fact, a detailed description of how architectural development is used to evolve this level of the model will be deferred to later sections that describe the evolution of methodologies to an engineering based paradigm. The basics though are the same as previously discussed. Business software systems, like other software systems, can be described in terms of specific architectures with associated standard component sets for implementation. The software development level of the information engineering process model is changed from one with a basic concern of creating custom systems for an enterprise to one concerned with matching enterprise needs to established architectures.

Enterprise modeling can be performed based on an engineering paradigm as well.

Enterprise modeling is a special case of domain analysis. Identifying classifications for

domains that describe enterprises facilitates the establishment of domain models in a similar fashion that architecture models can be established. Mapping enterprises to domain models would allow analysts to capture commonalities between enterprises and establish a foundation of principles and knowledge for enterprise modeling.

Once again, our goal is to adapt the process model to support routine engineering activity. Therefore, our same set of assumptions that were applied to the waterfall and spiral model will apply to information engineering.

3.6 Summary of Software Development Process Evolution

We have shown how the major software development process models have evolved from a common ancestry. Each model has been developed to address one or more specific issues that have confronted the development of systems of ever increasing size and complexity. We have also shown how the models can continue to evolve from creation based development to support the development of software solutions in a routine engineering fashion. The underlying commonalities of these models suggest the existence of some important facts with regard to adopting a true engineering paradigm.

Regardless of application domain, there exists fundamental commonality in the processes used to develop software systems. This is evident in the fact that all modern software development process models have derived from a common ancestor and by the fact that these process models can be described by a common meta model. This observation is important because it allows us to entertain the notion that there exists a fundamental set of software engineering principles that are applicable across all domains. Software architectural development is one example of a common software engineering principle.

The importance of this observation is that it supports the notion that a predictable set of properties along with standard processes for their use can be established as the foundation for a software engineering discipline. This supports the feasibility of defining standard architectures and related component sets as the foundation of software development.

This observation also encourages a reevaluation of the current trends to propagate specialized development process models and methodologies that are domain specific. Each process model or methodology should be considered as a tool in a collection of tools that may be used as appropriate in any domain. The selection of tools is based on the appropriate application of fundamental engineering principles.

We have now established how routine software engineering could be supported at the process model level. We now present a technique for adapting the methods used for actual implementation of these process models.

4.0 Software Architectural Development and Patterns

The basic strategy for evolving to routine construction of recurring systems is based on the adoption of software architectural development. It has already been shown through the previous review of development processes how the use of this technique effects analysis, design, and coding activities.

Methodologies provide definition of detailed activities that are used to actually perform analysis, design, and coding. New techniques will need to be introduced into these detailed activities in order to implement software architectural development. These techniques must support:

- representation of common architectures
- mapping of requirements to architectures
- mapping of architectures to software components
- integration of components via architectural structure
- adaptation of standard components to unique system requirements

The inclusion of software patterns as an integral part of standard software methodologies provides a means of including the necessary techniques to support architectural development. Patterns can provide a means of capturing and relating architectures and components necessary for performing software architectural development activity.

4.1 Implementing Architectural Development Through Patterns

In chapter 2 we addressed the definition of a software architecture. It becomes self evident from the description in chapter 2 of software architecture that the proper definition of a system's architecture takes more than a graphical depiction of connected components. Proper definition of architectures will provide the information necessary to evaluate system requirements against potential solutions. Our goal is to identify a technique that

will capture software architecture definitions in a way that facilitate their use in routine software development. Software patterns can be used to capture architectures as well provide a basis for a system that enables architecture selection, mappings to related software component sets, and guidance for the adaptation of standard components to satisfy unique system requirements.

We will adopt as our general definition of a pattern the definition offered by Riehle et. al. [RDZH96]. “A pattern is the abstraction from a concrete form which keeps recurring in specific non-arbitrary contexts.” This general definition is preferred because it does not restrict patterns to be only associated with software design artifacts which has been the primary focus of the use of patterns. Both Riehle and Zullighoven. [RZ96] and Buschmann et. al. [BMRSS96] present a case supporting the use of patterns to represent more than software design with patterns. Table 3 outlines the two pattern classifications schemes presented in these two works.

A review of table 3 points out the close correlation between the two classifications. Riehle and Zullighoven focused on the language used to describe a pattern where Buschmann et. al. focus on the solution the pattern provides. It is appropriate, and desirable, to describe architectural patterns with conceptual terminology, design patterns with design terminology, and idiom patterns with programming terminology. By correlating these two classification schemes, we are provided with a guide for the selection of proper pattern description from Riehle & Zullighoven for the appropriate solution category provided by Buschmann et. al.

4.1.1 Architectural Patterns

Software systems have always had architectures. However, only recently have there been efforts to identify these architectures and leverage their existence [PW92]. Architectural patterns provide a means of establishing templates for concrete software architectures that can be reused in successive developed systems. They express fundamental structural schemas and provide a set of predefined subsystems, specify their responsibilities, and

include rules and guidelines for organizing the relationships between them [BMRSS96]. The use of domain terminology in defining architectural patterns enhances the ability to map requirements to specific patterns which simplifies pattern selection.

Patterns of this type specify system-wide structural properties. The use of descriptive languages derived from terms and concepts of the domain relate architectural system structure to system requirements.

Riehle & Zullighoven		Buschmann et. al.	
Category	Description	Category	Description
Conceptual	A pattern whose form is described by means of the terms and concepts from an application domain.	Architectural	A pattern expressing a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, specifies their relationships, and includes rules and guidelines for organizing the relationships between them.
Design	A pattern whose form is described by means of software design constructs.	Design	A pattern that provides a scheme for refining the subsystems or components of a software system or the relationships between them.
Programming	A pattern whose form is described by means of programming language constructs.	Idiom	A low-level pattern specific to a programming language describing how to implement particular aspects of components or the relationships between them.

Table 3: Comparison of Pattern Categorizations

4.1.2 Design Patterns

Design patterns represent the middle tier of software structure. These patterns are typically used to represent recurring software components. The components are

represented independently of language specific implementation. Descriptions of these patterns is in terms of software constructs as opposed to domain specific language.

4.1.3 Idiom Patterns

Idiom patterns represent low-level programming language specific implementation of components or the programming language specific implementation of component relationships. These patterns capture useful recurring language structures.

4.2 General System Construction Based on Patterns

Figure 6 illustrates the general construction problem faced by software engineers. Comprehension on what is needed in a software system resides in a domain centric description. This description must be translated into software construction terms and models that support physical implementation of the software system.



Figure 6: General Software Construction Problem

Figure 7 shows how patterns are used to bridge the gaps between domain centric descriptions, software construction models, and physical implementation. System specifications are entirely within the domain description. Architectural patterns share the descriptive terminology with the domain and also contain software construction features thus bridging between domain description and software design. The Design patterns fall totally within the software construction description. The Idiom patterns are used to bridge from the software construction models into a physical implementation of the system.

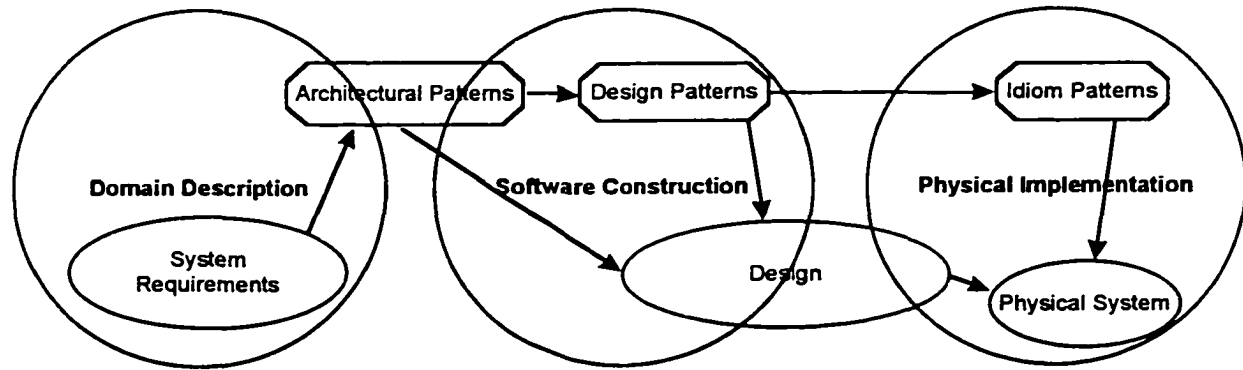


Figure 7: Pattern Use in System Construction

4.3 Pattern Format and Content

It is generally agreed that patterns contain:

- **Identity:** a name that uniquely identifies a pattern and is used as a basis for vocabulary.
- **Context:** a situation giving rise to a problem.
- **Problem:** the recurring problem arising in that context.
- **Forces:** any aspect of the problem that should be considered when solving it.
- **Solution:** a proven resolution of the problem [BMRSS96] [Flower97].

Beyond this there is little agreement as to the exact form and content of a pattern. A review of published patterns will show that many pattern writers prefer a free form approach. Where free form may seem to provide greater expressiveness for pattern writing, there are major advantages to adopting a standard fix format for patterns. A fixed format for patterns would provide:

- assurance that all pertinent information has been captured
- sharing of patterns across a broader population
- support for automated identification and selection
- support for pattern comparison
- support for standardized engineering handbooks

- support for formal definition and component proofs

The intent then for adopting a pattern template is to enable the pattern to be readily understood, applied, and implemented correctly. Table 4 is the template proposed by Bushmann et. al. [BMRSS96] and will be used as a starting point for defining a standard pattern template. A full description of the template in table 4 can be found in Bushmann et. al. [BMRSS96].

Field	Description
Name	The name and a short summary of the pattern.
Also Known As	Other names for the pattern, if any are known.
Example	A real-world example demonstrating the existence of the problem and the need for the pattern.
Context	The situations in which the pattern may apply.
Problem	The problem the pattern addresses, including a discussion of the associated forces.
Solution	The fundamental solution principle underlying the pattern.
Structure	A detailed specification of the structural aspects of the pattern.
Dynamics	Typical scenarios describing the run-time behavior of the pattern.
Implementation	Guidelines for implementing the pattern.
Example Resolved	Discussion of any important aspects for resolving the example that are not yet covered in the Solution, Structure, Dynamics, and Implementation sections.
Variants	A brief description of variants or specialization of a pattern.
Known Uses	Examples of the use of the pattern, taken from existing systems.
Consequences	The benefits the pattern provides, and any potential liabilities.
See Also	References to patterns that solve similar problems, and to patterns that help us refine the pattern we are describing.

Table 4: Pattern Description Template

The following describes a proposed variation on the template in table 4 to adapt it for better support for software architectural development. Fields not mentioned are intended to be used as described in Bushmann et. al. [BMRSS96].

It is important to remember that patterns are drawn from existing systems. Therefore, it is reasonable to assume that the details needed to develop a pattern are available or can be derived from existing system artifacts. It is also important to understand that architectural, design, and idiom patterns use the same pattern template for their representation.

4.3.1 Context

The context must clearly identify the domain the pattern was originally drawn from. This is only a minor extension of the current definition of context. It is important to identify the domain since this provides the basis for interpreting the language used to describe the pattern.

It is assumed that a domain analysis has been performed with respect to the system that the pattern is being drawn from. The importance of the information captured in this field is the provision of definitions of terms and a reference point for interpretation of the other fields in the pattern.

4.3.2 Structure

The structure section, along with the dynamics field which will be discussed next, must adopt a more formal representation than is generally used in pattern definitions. This is necessary because these two fields are used jointly to capture the definition of architectures in architectural patterns. For architectural patterns, we rely on the research in architecture definition to supply the appropriate representation of architecture just as one relies on design modeling techniques in design patterns.

4.3.3 Dynamics

Once again this section will need to evolve to a more formal definition of the dynamic characteristics of the components described by the pattern. This change has more to do with degrees of accuracy than being necessary for currently using patterns. From an engineering perspective we would like to be able to determine exactly how a system and its component parts will function before construction. This will improve our ability to engineer reliable systems. Advances in scientific measurement of software components will drive our ability to improve this field of the pattern definition [BMB96].

4.3.4 Implementation

This section will be expanded to include actual references to components in a standard library. Through this section, we establish links to components that can be used for actual system construction or provide an alternate method to define architectures and design or extend the definition capabilities of the structure and dynamics sections. Components may still be adapted but usually this would be to incorporate unique requirements for this particular system.

4.4 Required Additions to the Pattern Template

Four additional sections are needed to adapt a pattern template that is suitable to support software architectural development. These sections are:

- Companion Patterns
- Preconditions
- Constraints
- Pattern Type

4.4.1 Companion Patterns

This section is a refinement of the “See Also” section. Patterns by their nature are used with other patterns to form complete systems. Therefore, it is desirable to capture concrete relations that exist between patterns. Pattern relations identified in this section are more concrete than those listed in the See Also section. The intent of this section is to list patterns that should always be used in conjunction with a specific pattern unless the functionality provided by the underlying components associated with a pattern is not needed.

4.4.2 Preconditions

Preconditions for patterns is suggested by Beck [Beck94]. This section further refines our understanding of the relationships between patterns. Often the sequence in which patterns are considered is critical to their appropriate use and implementation. This section identifies for a given pattern the conditions that must exist before the pattern can be properly applied. These preconditions are satisfied through the use of related patterns. Therefore, this field serves as a link between patterns and a means of defining their sequence of use.

4.4.3 Constraints

Constraints for patterns is another suggestion by Beck, [Beck94]. Conflicting forces acting on a solution to a problem can exist and are often mutually exclusive. Typical examples of such constraints are choices between execution time and execution space, or development time and program complexity. A clear statement of these constraints aids in pattern selection and the predetermination of system cost, development time, and performance.

4.4.4 Pattern Type

The same pattern template is used for all three types of patterns. Therefore, it is appropriate to tag each pattern with its pattern type (architectural, design, idiom). Having this information imbedded in the pattern definition is additional information that complements the companion patterns and preconditions sections.

4.5 A Pattern Template that Supports Architectural Development

Field	Description
<i>Name</i>	The name and a short summary of the pattern.
<i>Pattern Type</i>	Architectural, Design, or Idiom
Also Known As	Other names for the pattern, if any are known.
Example	A real-world example demonstrating the existence of the problem and the need for the pattern.
<i>Context</i>	The situations in which the pattern may apply that include identification of the domain.
<i>Problem</i>	The problem the pattern addresses, including a discussion of the associated forces.
<i>Solution</i>	The fundamental solution principle underlying the pattern.
<i>Structure</i>	A detailed specification of the structural aspects of the pattern.
<i>Dynamics</i>	Concise description of the run-time behavior of the pattern.
<i>Implementation</i>	References to standard components used to implement the pattern.
Example Resolved	Discussion of any important aspects for resolving the example that are not yet covered in the Solution, Structure, Dynamics, and Implementation sections.
Variants	A brief description of variants or specialization of a pattern.
Known Uses	Examples of the use of the pattern, taken from existing systems.
Consequences	The benefits the pattern provides, and any potential liabilities.
<i>Preconditions</i>	Other patterns that must be satisfied before this pattern may be used.
<i>Companion Patterns</i>	Other patterns used in conjunction with this pattern.
See Also	References to patterns that solve similar problems, and to patterns that help us refine the pattern we are describing.
<i>Constraints</i>	Conflicting forces acting on the solution to the problem.

Table 5: Pattern Template Supporting Software Architectural Development

Table 5 shows the altered template that captures the information necessary to support software architectural development. Items added to or changed in the original template are marked in bold. Items that are considered crucial for supporting software architectural development are italicized.

This template provides the information necessary to support software architectural development. We use the information captured by the pattern template to develop a software development system based upon software architectural development. Figure 8 represents this development system. Domain needs are matched to an architecture through matching a system specification to the problem section in a set of related architectural patterns. This matching returns a proposed solution from the solution section of the selected patterns. The companion patterns field in the architectural patterns relate associated architectural patterns and the design patterns used to move development towards implementation. The preconditions section in the related patterns provide the information needed for proper coupling of patterns.

The companion patterns section in the design patterns relate associated design patterns and idiom patterns. The preconditions field in the related patterns provide the coupling information at the design level. At all levels, the implementation field links each pattern to standard components used as the basis for system implementation.

The other sections in the patterns are used in pattern selection and in support during implementation activity. Specific use of these fields varies as to the level of abstraction represented by the pattern.

By leveraging the abstractions of patterns as described, we provide an interface between domain specifications and system development activity. The patterns also help manage system complexity in two ways. First they provide an abstract vocabulary through pattern names that make it easier for engineers to discuss architectures without referring to specific details. Second they focus attention on a specific set of design and implementation options proven to be valid for a specific problem.

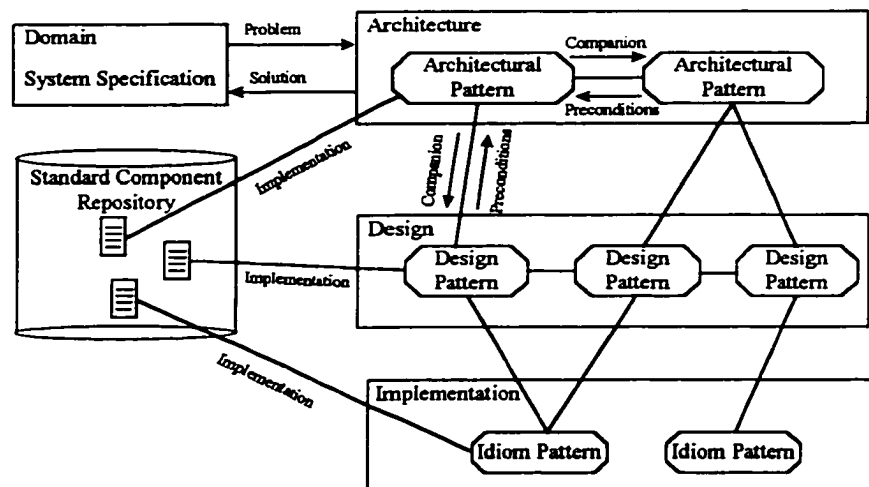


Figure 8: Pattern Support for Software Architectural Development

Chapter 5 will demonstrate how this generic development system can be used to evolve methodologies in current use today toward support for routine software development activity. To illustrate the use of this system and show that it is applicable to any domain, we will demonstrate how it can be blended with Fusion, a general object oriented methodology, and Rapid Application Development (RAD), a business systems development methodology based on information engineering.

5.0 Making Software Development Methodologies Engineering Based

Chapter 4 presented a general technique for software development based on software architectural development. It is not a complete nor specific methodology for software development. Many good methodologies have been developed over the years that address major issues in software development and support the established process models. Our goal is to evolve these methodologies using the technique described in chapter 4 so that they are engineering based as opposed to creational based methodologies. By doing so, we preserve the strengths provided by these methodologies while allowing software development to move towards a true engineering discipline. By evolving these methodologies in this fashion, we provide improved support for the routine development of common software systems.

If software development process models need to evolve to an engineering based paradigm, then so do the methodologies used to implement them. Just as there are commonalties between software process models, so are there in methodologies. Each new methodology created builds upon the experience of past methodologies and is created to address one or more specific issues in software development.

Methodologies, like processes, are creational based. This is not surprising considering that the same forces influencing the development of process models influence methodologies as well. We must first understand the issues being addressed by a specific methodology and its support for the process models that it implements before evolving the methodology to an engineering based paradigm. The goal of evolution activity is to preserve the basic characteristics of the methodology while shifting the underlying paradigm.

There are too many methodologies in existence today to address each one. Two have been chosen that will serve to illustrate the paradigm shift. Fusion, and Rapid Application

Development (RAD) represent the major methodologies in use today and provide a cross section of methodologies across the major application domains.

5.1 Evolving Fusion with Software Architectural Development

Many good object oriented based methodologies have been developed. We have chosen Fusion to represent this family of methodologies because it is itself a composite of several object oriented methodologies and it is defined by a very clear process.

Fusion was developed specifically to provide:

- a systematic process supporting team management
- well defined notations to aid team communication
- coverage of the entire software development life cycle [Coleman94].

5.1.1 General Description of Fusion

The Fusion method has combined features from several object oriented development methodologies into an integrated set of models that provide a direct route from requirements to a programming-language implementation. Underlying the models is a data dictionary used to define all the entities in the various models.

Fusion is more than a set of models. It provides a well defined process for using the models to develop a software system. The Fusion process is divided into three phases:

- analysis
- design
- implementation.

5.1.1.1 Analysis

The intended behavior of the system is defined in the analysis phase. The models used in this phase describe:

- classes of objects that exist in the system

- relationships between those classes
- operations that can be performed on the system
- allowable sequences of operations.

The models used in this phase are the:

- **Object model:** used to define the classes that exist in the domain
- **System object model:** used to define the system boundary within the domain
- **Interface model:** used to define system operations and their sequence of execution.

5.1.1.2 Design

In this phase, decisions are made as to how system operations are to be implemented by the run-time behavior of interacting objects. Operations are attached to classes, object referencing is defined, and class inheritance is established. The design models show:

- how system operations are implemented by interacting objects
- how classes refer one to another and how they are related by inheritance
- attributes of, and operations on, classes.

The models used in this phase are:

- **object interaction graphs:** describe how objects interact at run-time
- **visibility graphs:** describe object communication paths
- **class descriptions:** specifies the class interface, data attributes, object reference attributes, and method signatures
- **inheritance graphs:** describe class/subclass inheritance structures.

5.1.1.3 Implementation

This phase is where the design is turned into code in a particular programming language. Fusion provides guidance for this activity that provides direct support for using an object oriented programming language like C++ or Eiffel.

Fusion provides the following guidelines:

- Inheritance, reference, and class attributes defined in the class descriptions are implemented in programming language classes.
- Object interactions defined in the object interaction graphs are encoded as methods belonging to a selected class.

The permitted sequences of operations defined in the interface model are recognized by state machines. Figure 9 [Coleman94] illustrates how the various models work together to develop a system.

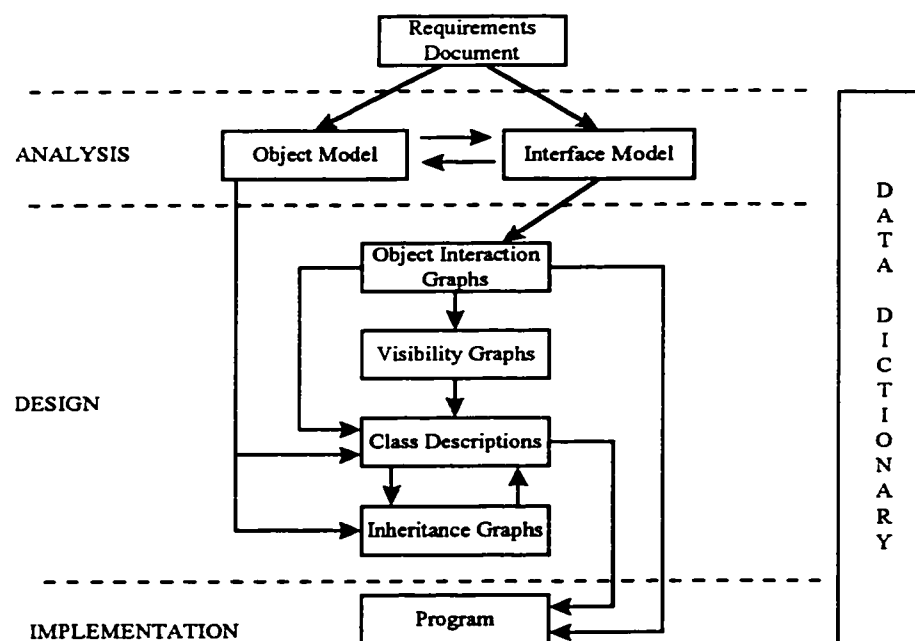


Figure 9: Fusion Model Relationships

5.1.2 Fusion and the Software Development Process

It is stated in Coleman et. al. [Coleman94] that Fusion must fit into the wider context of a software development process. The process in figure 10 [Coleman94] is an idealized model for software product production. The process follows the following stages:

1. Establish a stable product definition and set of requirements.
2. Define the major system hardware and software components.
3. Produce a plan for developing the system.
4. Develop each component identified in the system architecture.
5. Deliver the completed system to the customer.

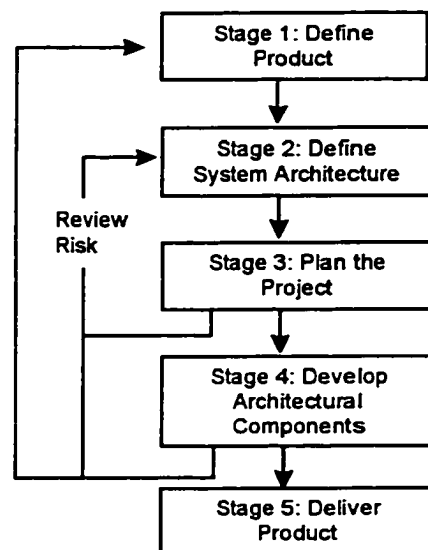


Figure 10: Process for Developing Software Products

This process and its implementation is based on the waterfall model and is enacted by incrementally developing the different architectural components. Fusion is seen as a methodology for implementing stage 4 of this model.

Referring back to table 1, we see that Fusion fits under the “Create Custom Models” and “Create Custom Components” phases of the creational meta model. By evolving to an engineering based process, the work of defining major software components found in stage two of the process model shown in figure 10 blend with activity in stage 4. This is due to the fact that custom models are no longer developed. Instead, established architectures will be identified as a final step in defining the system architecture which in turn will provide the initial system object models currently developed in the analysis phase

of Fusion. The following is the evolved definition of the stages of the process model in figure 10:

1. Establish a stable product definition and set of requirements.
2. Match the product definition to one or more standard architectural models
3. Produce a plan for developing the system.
4. Employ Fusion to refine the models associated with the standard architecture and adapt the associated standard components for this specific product.
5. Deliver the completed system to the customer.

5.1.3 Software Architectural Development Based Fusion

Standard architectures that are related to the product definitions developed in stage two are identified by referencing the context and problem sections of the architectural patterns. Selection of an architecture provides the foundation for the system object model through structure fields of the patterns that define the architecture. The dynamics and precondition fields provide the foundation information for the interface model.

The companion patterns fields of the architectural patterns are used to identify patterns that support the more detailed design of the product. In turn, the design pattern's structure and dynamics fields are used in conjunction with the established system object model and interface model to develop the object interaction graphs, visibility graphs, class descriptions, and inheritance graphs.

Once again the companion patterns in the design patterns are used to identify idiom patterns that will support final implementation. These patterns would primarily be used to aid in the coding of the class descriptions. The system life cycle developed in the analysis phase would still be the source of system state definition and the basis for implementing object interaction control.

5.1.4 An Example of Engineering Based Fusion

To illustrate the evolution of Fusion, we will use the case study contained in chapter 6 of Coleman et. al. [Coleman94] as an example. The requirements for this system have been reprinted in appendix A of this document. We will follow the modified development process stages showing how the system is developed. The modified Fusion methodology is used in stage 4

Stage 1: Establish a stable product definition and set of requirements.

The primary concern in this stage is to gather enough information to define the domain, identify required system functionality, and that allows the boundaries for the system to be defined. This stage is not directly addressed by Fusion and would be best served by the application of domain analysis techniques. In our example, the product definition has been supplied through the statement of requirements in appendix A.

Stage 2: Match the product definition to one or more standard architectural models.

It is at this stage where there begins to be a significant difference between creational and engineering based methodologies. In a creational based methodology, such as Fusion, a model would be created based on product knowledge showing the major system subcomponents and the high level interface between these components. In an engineering based methodology, the product can be recognized as belonging to a generic class of products which have defined standard architectures proven over time as being suitable for product development. We identify these architectures through patterns. The steps to this stage are as follows:

Step 1: Analyze the product definition to determine the generic class of the product.

- Identify the terms that capture the fundamental characteristics of the product.

- Step 2:** Identify a set of architectures proven suitable for this generic product class.
- Use the identified terms to search the architectural patterns using the context and problem fields.
- Step 3:** Assess each architecture to determine a best fit for this specific product.
- Perform a technical analysis using the structure and dynamics fields with supporting information from the implementation, example, and example resolved fields to determine applicability.

Consider for a moment a simple requirement to connect two points on opposite sides of a river. A quick analysis of this requirement suggests that the product required is of the generic class bridge. Bridges have many different types of architecture. A bridge may be a suspension, cable, or draw bridge. The engineer will identify how the bridge is to be used along with the physical characteristics of the river banks and length of the expanse across the river to determine the best bridge architecture. The principle is the same for this stage of software development.

In our example we have a requirement for a petrol purchasing system. If we abstract out the fact that the merchandise being sold is petrol, we can identify the system as belonging to the generic class of sale and transaction systems. Assuming the existence of a standard architecture library, we would find several different architectures for this generic class. There would be architectures for point of sale systems, telemarketing sales systems, remote location sales systems, cash only sales systems, and so on. The first task of the software engineer would be to select from the set of standard architectures for sale and transaction systems an architecture that best fits the requirements for this particular product. The standard architecture will supply the fundamental high level design model for the system.

In our example, by realizing that the requirements describe a sale and transaction system that employs a customer operated dispensing unit that is monitored by an attendant responsible for processing payments, we could search our patterns library for architectural

patterns that have problem description that relate to this type of system. Appendix B contains an example of a pattern that might be selected. This pattern represents a very high level general structure of a potential solution. The companion patterns section would point to related architectural patterns that would be used to complete the detail of the system.

How much work is required in this stage depends on the following:

1. Determination of a generic class for the product may not be straight forward. Poorly written requirements or descriptions written in detail domain language may make it difficult to identify the generic characteristics of the product.
2. The underlying pattern definitions may not describe the architecture in terms or format that readily translates into Fusion models. The structure fields in the architectural patterns will provide the information necessary to construct the initial system object model. If these structures are defined in the patterns using techniques that are similar or the same as the methodology used to design the product, there would be considerable reduction in the effort required to build the initial design models.

Figure 11 represents a possible generic system object model obtained from the selected architectural patterns that might be retrieved from a properly populated library of standard architectures. Since such a library does not currently exist, this model was derived from the base example in [Coleman94] by abstracting away all the domain specific details of the system object model. In this generic state, this system object model could serve as a starting point for many different sale and transaction systems.

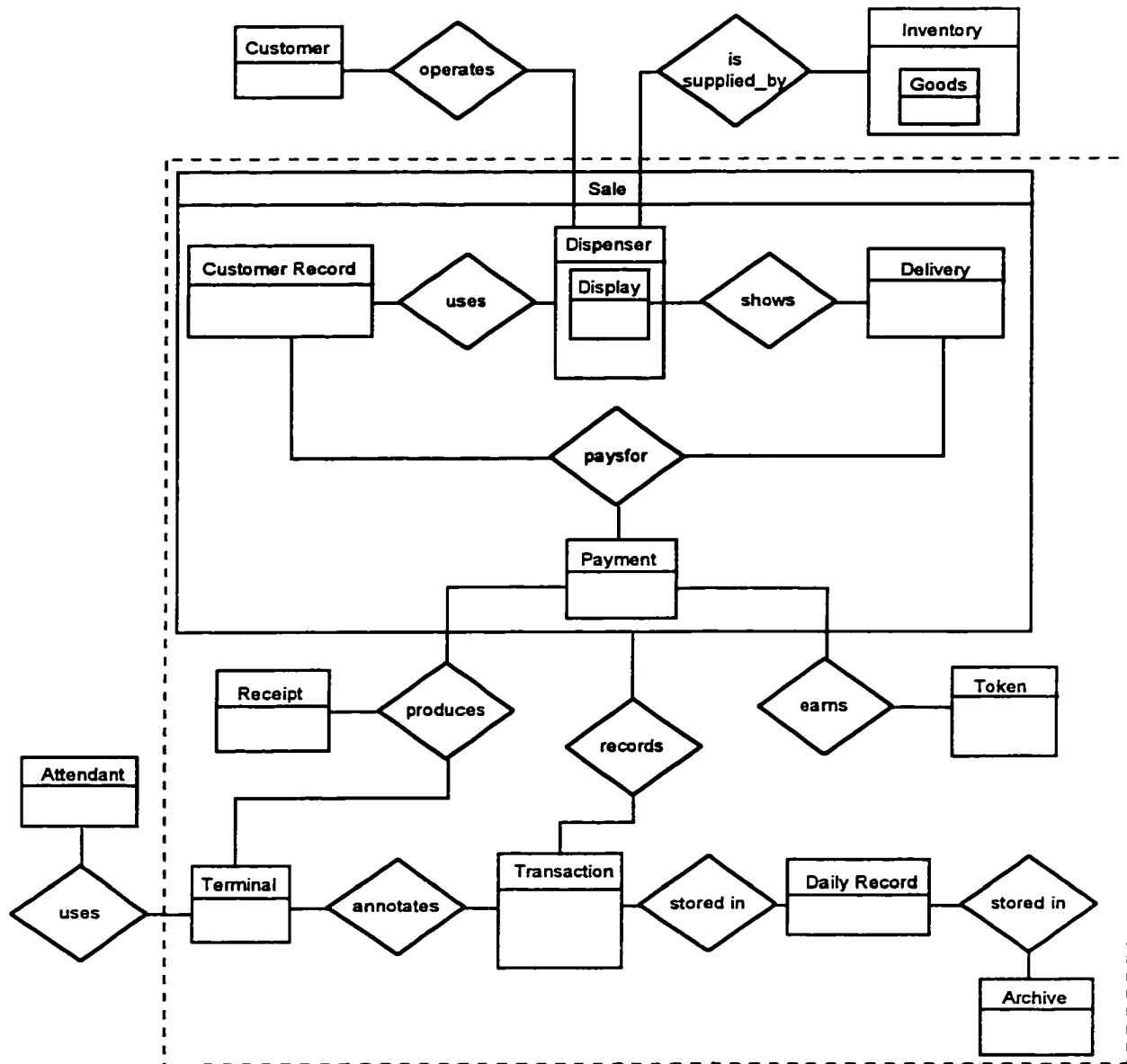


Figure 11: Sale and Transaction Generic System Object Model

It is reasonable to assume that the model in figure 11 would be a composite of several patterns. The relationships between patterns would be documented within the pattern definitions. Providing standard architectures as composites of patterns enhances the flexibility required to both adapt architectures for specific systems as well as enhancing the ability to apply the same architecture to many different domains and similar systems.

Stage 3: Produce a plan for developing the system.

A plan is developed in this stage that defines the activity that must be performed to complete the architecture and develop the components of the system. A vital part of planning is the identification of project risk. Basing system development on a standard architecture provides a firm foundation for project planning activity.

The generic architecture provides a clear understanding of the high level components needed for the system. The architecture should already identify all major system components. Furthermore, information in the associated patterns will lend insight into the use of the architecture and potential areas of risk.

General system development risk is reduced by use of a standard architecture since it is based on approaches already proven. Considerable information that is not available in traditional creational development will be captured in the pattern definitions which will increase understanding about the system which in turn reduces risk.

Stage 4: Employ Fusion to refine the models associated with the standard architecture and adapt the associated standard components for this specific product.

Fusion addresses the activities of this stage in the development process. The phases of Fusion are described Coleman et. al. [Coleman94] as a set of steps. We will now describe how these steps are modified when Fusion is based on an engineering paradigm.

Analysis Phase

The primary purpose of this phase in Fusion is to create the fundamental system models that subsequent development will be based upon. This phase will experience the most impact from the adoption of software architectural development since the use of standard architectural models reduces or eliminates the need to create base system models.

Analysis Phase Step 1: Develop the Object Model

The purpose of this step is to capture the concepts that exist in the domain of the problem and the relationships between them. It entails brainstorming a candidate list of classes and relationships, establishing a data dictionary, and constructing an object model.

Much of the work in this step is eliminated by the use of a standard architecture model for the product. Generic classes and their relationships are provided by the architecture. The work in this step becomes one of review and adaptation of the standard architecture.

There may also be the need to translate the architecture into a Fusion object model if the pattern's structure is not already expressed as one. We would then adapt the generic classes represented in the architecture to the unique features of the system. In this case, the Class Inventory becomes Storage Tank and the Goods contained within the Inventory becomes Petrol, Figure 12. The Dispenser class also becomes Pump. Attributes unique to this system should also be added to the classes at this time.

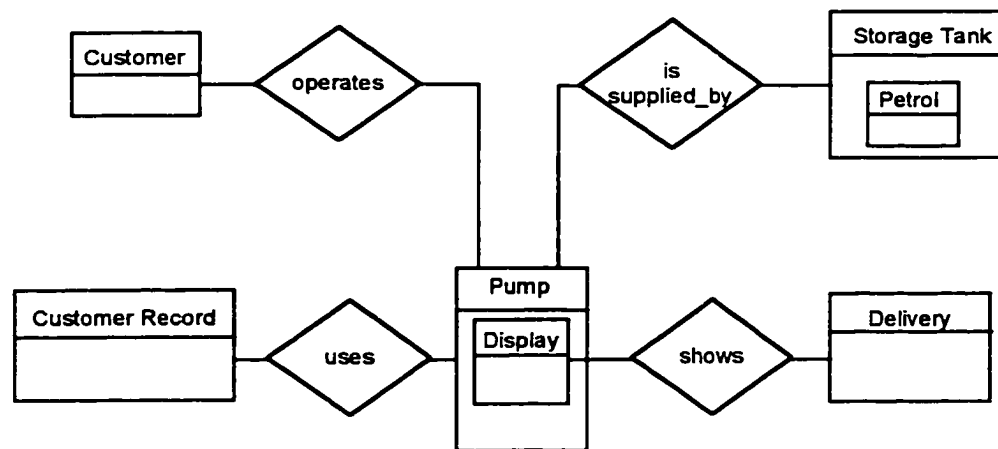


Figure 12: Standard Classes Adapted for Unique System

Further definition of unique classes may also be needed if software is to be developed to control a specific object represented by a class. For instance, software may be needed to control the pump that will supply the petrol. If these classes represent entities that routinely occur in systems, then there should be standard architectures for those entities. Since we are using a standard development methodology, they can always be created if they do not represent routine design issues.

Analysis Phase Step 2: Determine the System Interface

This step identifies the set of system operations agents external to the system that will initiate system events which will produce system output in response. This activity defines the boundary for the system. The primary output of this step is the system object model. This model is a refinement of the object model in that it identifies which classes in the model will be implemented within the system and which classes represent agents external to the system to be developed.

The context, problem, and solution sections of the patterns that define the standard architecture directly address the identification of system operations and system boundary. As such, the work usually required in this step is generally eliminated. The definitions provided by the patterns must be reviewed against the requirements to assure that the requirements will be satisfied. It is expected that some adaptation of the standard operations will be required and would be added in this step.

In our example, the system interface is defined as part of the standard architecture. This defined interface must be reviewed after the object model has been adapted for specific requirements to assure it is still appropriate.

Analysis Phase Step 3: Develop an Interface Model

The interface model is composed of two distinct yet related models which are generated in this Fusion step. The life-cycle model and operation model are defined in this step. The life-cycle model defines the allowable sequences of system interactions. It defines what operations respond to specific system events and in what order. The operation model defines the semantics of each system operation in the system interface. A specific template is used to document each operation in the operation model.

Scenarios are used by Bushmann et. al. [Bushmann96] to assist in the documentation of the dynamics of a pattern. Coleman et. al. [Coleman94] recommends that scenarios be developed to assist in the development of the interface model. We could enhance the ability to capture the dynamic behavior of patterns by adopting a more formal approach to

dynamic definition such as the one used in Fusion. There already exists a common base through the use of scenarios. Patterns, as they exist today, must rely on free form text to supplement the information that cannot be captured in a scenario such as alternative paths.

The adoption of the Fusion interface model as the basis for pattern dynamic description would also allow for more precision. This added precision supports the use of measurement and formal methods for pattern selection as well as a basis for building system models that help predict system performance. All this makes a strong case for adopting a more formal representation for dynamic behavior definition such as the Fusion interface model.

To illustrate the expressive power of the Fusion interface model consider the following. Lets assume that an architectural pattern named Transaction is part of the standard architecture. In the dynamics section we would find a description of the behavior for accepting payment from a customer. Typically, it would be expressed using a scenario like the one in Figure 13. Scenarios are limited in what they can represent and tend to lack the degree of precision needed for a complete understanding of system dynamics Figure 14 represents customer payment using Fusion. It allows sequences of events to be clearly defined and provides a means to show alternative paths without the use of multiple scenarios.

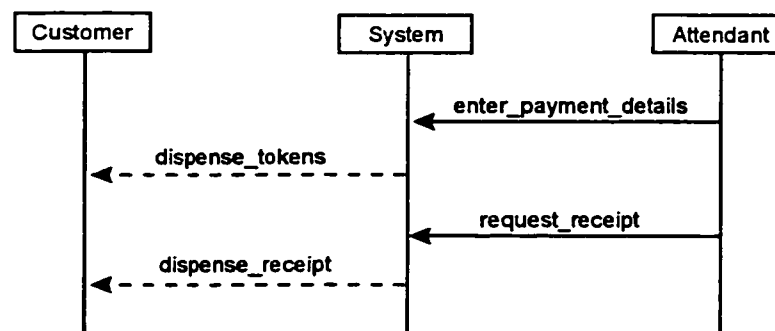


Figure 13: Scenario for Payment

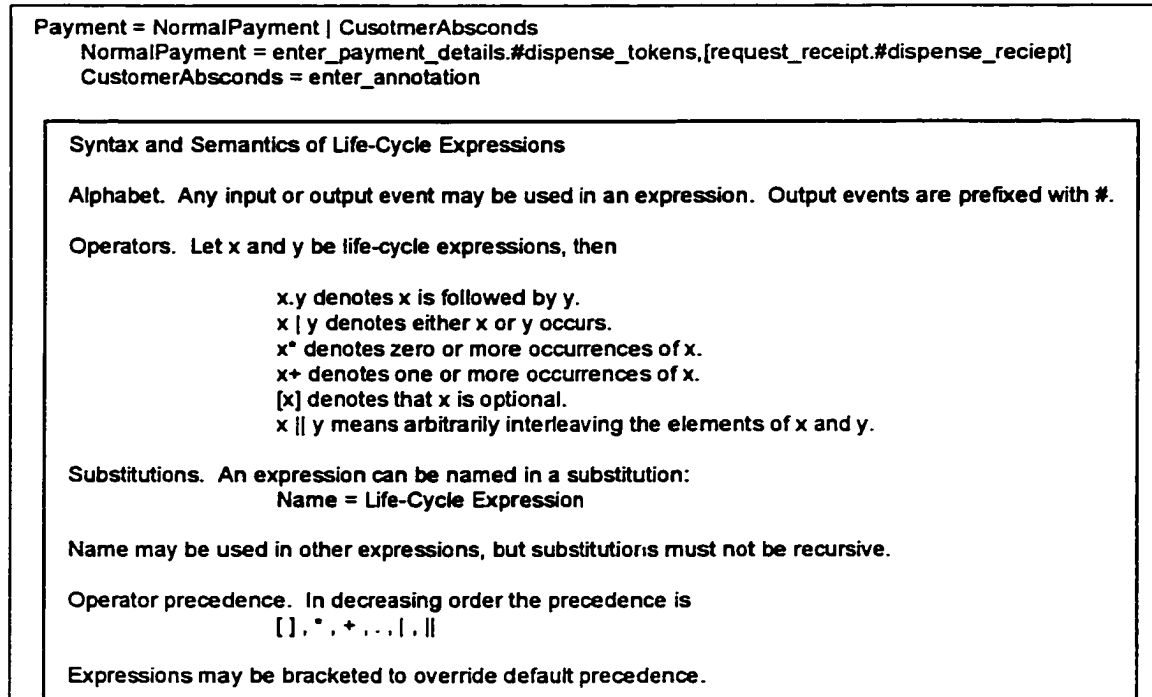


Figure 14: Life Cycle for Payment as a Regular Expression

As mentioned earlier, in Fusion, the dynamics of a system are defined by the life cycle model and its companion operation model. Together they are called the interface model. The operation model clearly defines the change of state and events that are output by system objects. The operation model provides a schemata that aids in capturing this information and simplifies the task of translating the dynamic definition into actual code. It would be appropriate to include the operation models as components in the standard component library and reference them in the pattern through the implementation section.

Analysis Phase Step 4: Check the Analysis Models

Completeness and consistency of the analysis models is checked in this step. This remains a necessary step and is unchanged by any modifications required to adopt an engineering paradigm. Reviews such as this play a major role in instilling discipline within the process and provide the visibility needed to manage the project as well as assure desired levels of product quality.

Design Phase

The design phase in Fusion, like any software development methodology, is where the abstract models of the domain and system requirements are transformed into software structures. The concern of design for Fusion is the identification of the objects which will be used to implement each identified system operation along with the communication protocol between the objects.

In an engineering based environment, we will recognize the routine aspects of a system and have available proven design components that can be used to satisfy system requirements. Software architectural development provides a means of identifying routine design components for specific systems through the tie between the standard architecture and the underlying design. This tie is maintained by utilizing the companion patterns section of the architectural patterns. A software engineer will find references in the companion patterns section to design patterns which have been used in the past to implement the system architecture. The design patterns referenced will contain the information necessary for the engineer to assess their usefulness and guidance for any needed modifications to allow for unique aspects of the system. The following will describe each Fusion design step as it would be implemented based on these principles.

Design Phase Step 1: Develop the object interaction graphs.

This step develops a set of graphs that define how the functionality for system operations is distributed across the objects of a system. It identifies the relevant objects involved, establishes the role of each object, defines the messages between objects, and records how the objects interact.

These graphs are concerned with the design of the dynamic behavior of the system. Therefore, we would expect to find information necessary for producing these graphs in the dynamics, implementation, preconditions, and possibly the companion patterns sections of the architectural and design patterns that define the standard system. Exactly which sections will be used and whether the information is contained in the architectural

patterns or a set of associated design patterns will depend upon the complexity of the system under development and the method used to document the patterns. More complex systems will require more decomposition in their description and will incorporate more patterns. Therefore, in this case it will not be possible to capture the level of detail needed to develop object interaction graphs in the dynamic sections of the architectural level patterns. This level of detail would be found in the associated design patterns referenced through the companion patterns section. The detail for smaller less complex systems might be captured in the dynamic descriptions of the architectural patterns.

If the patterns are documented using Fusion, then the interaction graphs would be found as part of the implementation description for the architectural and design patterns. If the pattern descriptions are of a more generic format, the software engineer will be required to transform the dynamic information into Fusion notation.

Regardless of the complexity or format of the patterns the activities in this step remains the same. Object interaction graphs are established for the standard architecture. This provides a generic model which can then be modified to include the unique specifications for this specific instance of the software product under development.

Design Phase Step 2: Define a visibility graph for each class.

Up to this point it has been assumed that all objects are mutually visible and can send messages to each other. In this step object visibility is defined in detail. Identification for each class is made of the objects that need access and the appropriate kinds of reference to those objects. Each message in an interaction graph is inspected and a decision is made as to the kind of visibility reference required based on the lifetime of the reference and the target objects visibility, lifetime, and mutability.

Although generic designs may suggest object visibility, the unique characteristics of a specific instance of a product will heavily influence what the final object visibility should be. Therefore, this step in Fusion remains essentially unchanged.

Design Phase Step 3: Develop class descriptions

The class descriptions specify the internal state and external interface required by each class. They serve as the basic specification for coding. Class descriptions record:

- methods and parameters
- data attributes
- object attributes
- Inheritance information

Generic class descriptions should exist in the standard component library. They are referenced by the patterns used in system design through the implementation section. The primary task in this step is to modify the data and object attributes. Data attributes are modified to include unique characteristics of the product. Object attributes are modified to reflect the visibility definitions established in the visibility graphs.

Design Phase Step 4: Develop inheritance graphs

This step is used in creational based Fusion to review the objects and classes and define superclasses and subclasses. This step should not be necessary following an engineering based methodology since the basic structure of the system is previously defined through the use of routine architectures.

Design Phase Step 5: Update class descriptions

In creational based Fusion, the inheritance information developed in step 4 must be included in the class descriptions. This step will also be eliminated in an engineering based methodology for the same reason cited in step 4.

Implementation Phase

The two primary strengths of Fusion is its well defined process and the flow from one model to another all the way through implementation. Because of the strength of Fusion's set of models, we can consider two alternatives to engineering based implementation. The

first alternative follows closely to conventional code reuse techniques. The implementation sections of the design patterns could reference code segments that would implement the design in a generic fashion. The code could then be modified to fit the unique needs of the product. The second alternative is to write the code based on the system life cycle models and class descriptions referencing idiom patterns to support development of method bodies.

We can consider the second alternative because development of the actual code from these models is fairly straightforward since the majority of the complex design decisions have already been made and the structure of the class descriptions follows closely the component structure used in object oriented programming languages. Additionally, coding activity expends a relatively small amount of the total system development cost [Boehm81]. Therefore, the cost of maintaining and using code components in a standard library may do little to reduce overall system development cost.

Relying on standard code components may also preclude the use of fourth generation languages and automated code generators. We would also be required to maintain several different code components for the same design to accommodate different platforms and languages.

By maintaining class descriptions and idiom patterns as the base component parts, we increase the ability to apply the architecture and its associated standard components to many different domains and implementation platforms and languages. Idiom patterns, which are language dependent, provide the necessary guidance to develop code segments not directly addressed by the class descriptions. These segments would address specific repeating functionality, error handling, and iteration within the method body.

5.1.5 Review of Engineering Based Fusion

Fusion provides a well structured approach to software development. We have shown that the creational basis of Fusion can be removed and replaced by architectural development techniques. This transformation retains all the benefits of Fusion while

enhancing the overall methodology. In the evolved methodology, we have eliminated the brainstorming and iterative discovery common in analysis and compressed the overall life cycle. We have also provided a means of basing development on proven software components (architectures, design, code, test cases, etc.) which enhance the ability to cost and manage projects and reduce overall project risk. We have also shown that Fusion provides representations that can be incorporated in the pattern definition to enhance the pattern's descriptive capabilities.

5.2 Evolving Rapid Application Development with Software Architectural Development

Rapid Application Development (RAD) is a methodology created specifically for the development of information systems. It is designed to be used within the context of Information Engineering. It was created to specifically address the need to produce information systems in less time at lower cost while maintaining high product quality. The methodology relies heavily on the use of small development teams using advanced code generation tools based on a rich repository. Appendix C is an outline of the overall methodology. The outline is derived from the method charts found in [Martin91]. Not all the specifics were included in the outline. It is included to provide a reference framework for discussing how this methodology would evolve into an engineering based methodology.

RAD establishes a development team that incorporates the user as an active participant in all phases of development. The basic premise is to quickly develop system requirements and design with user support within the overall context of a pre-established business enterprise model that defines the overall business goals and information system needs for the organization. Requirements and design are then translated into a software system by a succession of prototypes created by small teams. These prototypes are developed in short development cycles that allow the user to review the product and provide feedback in a timely fashion. The primary justification for this approach is the realization that

requirements quickly become outdated for large systems when there is an attempt to completely define the requirements before developing the system.

Control of system development relies upon the use of CASE tools to enforce coding standards. Automatic code generators are also used to quickly develop data screens, reports and basic system functionality.

The RAD methodology is used to implement the System Planning and Design, and Construction and Cutover phases of Information Engineering, figure 3. As such, we will restrict our focus to these phases of the overall process.

5.2.1 Basic Need for Evolution to an Engineering Based Paradigm

The creator of the RAD methodology might well contend that this methodology is already based upon the engineering paradigm as it has been defined in this document and to a degree we would agree. The incorporation of a development repository for software artifacts along with aspects of the methodology that encourage developers to search for similar existing systems does move the methodology in the right direction.

However, there is nothing in the methodology that addresses the use of artifacts found in the repository within the context of defined standard architectures. The closest thing to this notion is the use of a master encyclopedia for the data models. The encyclopedia is used to assure that the various systems developed do not corrupt the overall data base for the organization.

What is needed then to completely establish this methodology as an engineering based methodology is to incorporate an activity that allows the repository artifacts to be used based on the identification and utilization of standard architectures.

5.2.3 Modifications to System Planning and Design Phase

The system planning and design phase is addressed by sections 3 and 4 in the RAD outline, appendix B. Requirements planning and design is accomplished by the use of two

well defined techniques called Joint Requirements Planning (JRP) and Joint Application Design (JAD). They are called “Joint” in that requirements and design are developed jointly with the user. The basic premise is that development is expedited when both software developers and the users who have authority to decide what the system is to do work together to produce the requirements specifications and top level design. Requirements and design are developed in workshops that drive the development team toward consensus on the final requirements specification and design. Emerging requirements and design are captured in a CASE tool that can be used to quickly create prototypes that help the team visualize the system. JRP and JAD may be conducted as two separate activities or as one depending on the complexity of the system to be developed.

Part of the preparation for a JRP or JAD workshop is to prepare an initial model of the requirements specification or design. The current description of RAD relies upon research of existing systems used by an organization and a general review of information maintained in the repository. Our approach to completing the evolution of these activities to an engineering based paradigm is to formalize this research based on the use of standard architectures and patterns as previously described.

The research step 3.2 in the outline becomes more aligned with domain analysis and the development of the system specification. The main bulk of this step is already concerned with discovering the fundamental information needed to define system functionality and to create a development plan. Sub steps 3.2.6, 3.2.7, and 3.2.8 would be removed from this step. These steps are used to create a tentative prototype of the developing system that is used in the JRP to help team members to visualize the system. It is also expected that the artifacts identified in these sub steps would become the basis for the new system.

What is desired in the requirements planning phase is the documentation of requirements in terms of the domain and the selection of a standard architecture as the basis for further design and implementation. Selecting artifacts from the repository without an established architecture is premature and could lead to adopting solutions too early in the

development cycle. The system specifications define the problem. This description is used to search for architectural patterns that have been used as solutions to the problem. A standard architecture can be identified through this search.

The selected standard architecture becomes the basis for the first JAD workshop. It will provide either the initial design diagrams or the information needed to create initial diagrams. What is provided by the architecture depends upon how the architecture is defined in the underlying patterns. These initial diagrams are created in sub step 4.1 as part of the workshop preparation. Having this foundation expedites the workshop. The main concern of the workshop is to identify the unique aspects of the system and adapt the architecture and fundamental design already provided by the standard components. It would be expected that the bulk of the effort expended in this step would be to define the user interface portion of the system. There would also be a need to review the architecture with respect to its interface with other systems within the overall enterprise.

5.2.4 Modification of the Construction and Cutover Phase

The requirements planning and design phase concludes once four types of design diagrams have been developed for the system. The entity-relationship diagram defines the logical design of the underlying data base. The decomposition diagram defines the major system processes and decomposes each process into its constituent functional components. The dependency diagram which is a data flow diagram shows data entity transformations through the functional components. The action diagram is a structured textual description of the functional components similar to pseudo code. The purpose of this phase is to translate the entity-relationship diagram into the physical data base structure and use the remaining three diagrams to develop the system code. Current information system development relies heavily upon tools that automatically create physical data base structures from logical models and the use of fourth generation languages to develop functional code. The existence of these tools provides support for the notion of only maintaining patterns in our standard components library. The effort required to create the physical representations of the design is small compared to the overall development effort

and does not justify maintaining large code libraries. The one exception would be commonly used standard functions that roughly correspond to idiom patterns. These types of functions are usually provided by the fourth generation language systems. Some of these tools, such as PowerBuilder, also provide a means for the developer to create user defined functions as well.

In light of this, there is little change needed to the steps supporting this phase. The only addition would be the use of idiom patterns to extend the breath of functions available to the user or to aid in the selection of the best approach to low level code structure.

5.2.5 An Example of Engineering Based RAD

The RAD methodology is intended to produce a product that is structurally different to the object oriented product produced by Fusion. The basic software architecture will be segmented along the general lines of data representation and functional representation.

Data representation or modeling is the predominant concern of this methodology. The primary goal in the software architecture will be to first capture the data entities and their relationships and then capture the processing of the data. A data entity is a meaningful related set of information that retains state over time. The relationships between entities imply that one entity takes action or maintains a certain state relative to another entity [RG94]. What is needed then from a basic system architecture is a data model for the system, identification of system functions or processes, and a representation of how the defined system processes interact with the data. Let us assume that we are using the same set of patterns described in the Fusion examples and that these patterns are described using Fusion. This immediately introduces the need to add a translation step for RAD based development. However, this does not introduce significant overhead to development and will serve to further illustrate the advantage of having standard architectures and component libraries based on patterns.

Referring back to Figure 11 we can derive the data model shown in figure 15. The concern of the data model is to capture the entities that will have persistent information

and identify the relations between them. The data model shows that a dispenser is related to many customer records while each customer record is only related to one dispenser and so on.

As shown, the data model is incomplete since the attributes for the entities are not defined. Attributes in a data model become the fields within the tables each entity represents. The attributes also carry a good deal of information regarding the use of the data model. For instance, the data model as shown does not indicate that the attendant can annotate the transaction record. The attribute “annotation” should be added to the transaction entity to capture the fact that transactions can be annotated. The method for entering the annotation is not captured in a data model. This highlights the primary difference between an object model and a data model. The information based architecture will remain more abstract than the object model in that it never truly captures the system structure in terms of real world objects. We must split data and functionality into two different models then show their relationship in a third.

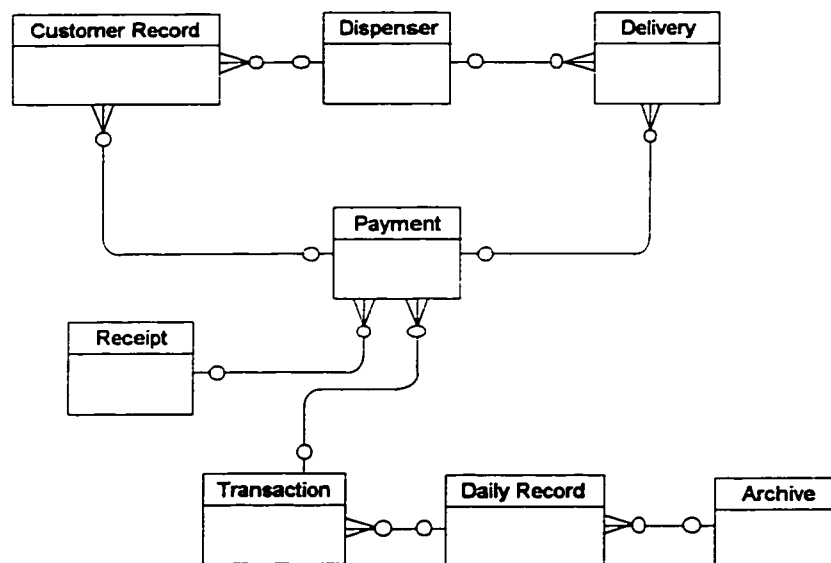


Figure 15: Data Model for Sale and Transaction Architecture

The next model to be built would be the process decomposition model. Once again we can derive this model from the system object model. The majority of the relationships recorded in the system object model represent system functionality. In addition, the

operation model is a source for system process definitions. To illustrate the process decomposition model consider the process “payment” shown in figure 14. Figure 16 shows how these functions would be represented in a process decomposition model. You will notice that all notion of allowable sequence is lost. This information would be captured in the action diagrams.

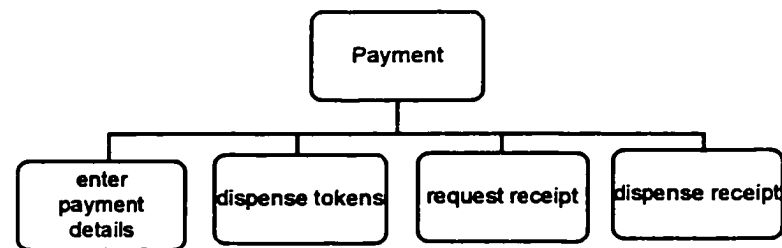


Figure 16: Process Decomposition Model for Payment

To complete the architecture we will need to create a data flow model that represents data entity transformation by the processes and an action diagram to capture the process flow of system dynamic activity. We will not develop this portion of the architecture since the examples already given are adequate to demonstrate the principle of architectural based development and the use of standard architectures in conjunction with standard component libraries populated by patterns.

5.2.6 Review of Engineering Based RAD

The primary change in this methodology is the inclusion of standard architectures as a basis for system development. The methodology already supports the use of a repository. However, the methodology is creational since the repository is only used to support the development of new models. We have also demonstrated that the patterns supporting a methodology need not be described using that methodology’s nomenclature. In particular, patterns expressed in terms of objects are readily translated into models traditionally used for information system development.

5.3 Summary of Methodology Evolution

In this chapter we have shown how the template pattern and general development technique based on software architectural development, described in chapter 4, can be used to evolve existing methodologies from a creational based paradigm to an engineering based paradigm. Patterns provide a template for the capture of information about components at the architectural, design, and code level. The patterns can then support a system whereby domain specific specifications can be related to software architectures, architectures to detail design components, and design components to code components. This system also supports the integration of a standard component repository to enhance the ability to reuse actual software components.

We have also demonstrated how this technique can be combined with existing methodologies, specifically Fusion and RAD. By combining our pattern based technique with existing methodologies, we enhance the methodology's ability to support routine repetitive software development while retaining the advantages the methodology was created to provide. By using Fusion and RAD as example methodologies, we have shown that this technique can be combined with any methodology regardless of software domain the methodology was developed to support. The inclusion of support for routine software development is essential to evolve methodologies out of creational based software development software engineering.

6.0 Conclusion

6.1 Existence of Scientific Basis to Support the Paradigm Shift

We have already stated that engineering is based on scientific discovery. Having defined what we believe to be a proper definition of software engineering and having described the evolution of basic process models and methodology to support that definition, we must still determine if there exists a sufficient scientific basis for the performance of engineering activity.

A key to determining the existence of a scientific basis is our ability to measure software components. Engineering disciplines rely on established systems of measurement and proven relationships among measures [Card88]. These measures originate from scientific discovery. If then there exists a system of measures capable of supporting development activity, we can conclude that a sufficient scientific basis exists to perform software engineering.

A quick survey of publications will provide a large listing of references on measures for software components. There are measures for complexity, coupling, cohesion, reliability, understandability, performance, quality, and so on. One can also find that these measures have been found to serve as a basis for software development. What one will not find is universal acceptance of the measures or broad application of the same measure. Metrics tend to be most effective in specific domains or within specific organizations.

So then the issue with software measures is not one of their existence but one concerning their precision and standardization across the industry. Lack of precision or standardization does not preclude the use of measures as support for software engineering. Indeed, the adoption of engineering discipline will improve measurements by providing structure and repeatability both in process and reuse of the products being measured.

Another key area of scientific support is the establishment of theory. Theories are needed in software engineering to establish search systems to correlate domain problem descriptions to architecture, establish architecture structures, define compatibility of design components, and establish program code structures to name only a few areas needing support.

Once again the issue is not one of the existence of theories but one of sophistication, formal definition, and wide spread applicability.

Computing science has provided the basis for the performance of software engineering. Our scientific basis may not be as precise as we wish nor are our theories as well defined as we would like. Nevertheless, development environments can and have been built on this body of science that support software development.

It is important to remember that the establishment of theory in engineering practice follows a recognizable pattern that often has a cycle time of up to twenty years. Most theories begin by recognizing repetitive successes as hoc solutions. Eventually these solutions are accepted informally as a set of folklore. As the folklore becomes more systematic, the solution is codified as a set of heuristics and rules of procedures. Eventually these codified procedures become crisp enough to support theories. These help to improve the practice which allows us to consider harder problems and thus start the cycle over again [SG96].

One of the advantages we would hope to see from the adoption of software architectural development is a shortening of the cycle time associated with theory development. We can expect this from the more formal definition of routine repeatable development activity. On the other hand, the lack of standardization across the industry itself may prove to be advantageous from a business standpoint since it provides an organization the opportunity to create a competitive advantage that is non-reproducible in other organizations. Where this may be a business advantage it will impede overall movement in the software industry towards true software engineering.

6.2 Related Work

Through out this work, we have cited several areas of research focused on the underlying technologies that are brought together in the implementation of software architectural development. The two most notable of these technologies are software patterns and software architectures. We were unable to find any work suggesting that the fundamental underlying paradigm of software engineering must be changed to advance software engineering practice. However, the goals of research in domain engineering are very closely related to approaches proposed in this thesis. These similarities are most clearly seen in the work by Peterson and Stanley [PS94] where they propose a technique for mapping domain models and software architectures to generic designs. Their work is based on the following two premises:

1. A domain model, the product of domain analysis, embodies the requirements for software in a domain.
2. Software architectures exist that provide a framework for generic designs.

Although not directly stated in this work or related work in domain engineering, there is a strong implication that software engineering should evolve to be based on the establishment of well understood domain models that map to defined architectures and designs. Similarly, we see this same unstated theme in work related to the use of software architectures [WL97]. This in principle is that same goal for software architectural development. We certainly see opportunity to incorporate the concepts of domain engineering with software architectural development. We also feel that although work in domain engineering does not specifically call for a change in the underlying paradigm of software engineering, it supports the need for a reevaluation of the current paradigm.

6.3 Summary

Software development can become a true engineering discipline. The greatest challenge to evolving software development from a craft to an engineering discipline is not technical

but mental. Software developers must begin to view software development as the routine application of software components to recurring problems based on established software architectures as opposed to seeing each project as the creation of a new design and program.

The process models and methodologies commonly used today are easily transformed to support this engineering paradigm. They are transformed by identifying those portions of the process or method that are creational and replacing them with techniques based on software architectural development.

Software engineering is also domain independent. From our application of software architectural development to both Fusion and RAD we have shown that there are underlying engineering principles that can be applied regardless of domain or type of system being developed. We have also shown that process models share a common heritage. This suggests that we should not think of software processes and methodologies as being domain specific but rather as defining generic processes and methods that are based on engineering principles that can be tailored for specific domains. By doing so, we will begin to realize that traditional barriers between domains such as business applications and scientific programming are artificial and begin to be able to benefit from advances in computing in industry at large.

6.4 Future Work

The obvious next step to this work is to create a working development environment based on the engineering paradigm and techniques described. To accomplish this we will have to combine elements from many related fields of research.

Patterns and pattern systems will need to be developed that support the mapping of architectures to domain oriented problems as well as relate design patterns to architectural patterns and code or idiom patterns to design patterns. This task will require combining elements of research from domain engineering, patterns, architecture languages, design modeling, and code abstraction techniques.

The patterns and standard software components will require a robust repository. Elements from research in repository data base design and software component reuse will be needed to implement this portion of the environment.

Appropriate standard architectures and other standard software components will need to be created. Techniques that have been developed to transform components in legacy systems into reusable components will be helpful in this activity.

Activity within the environment must be managed. Process activity and products produced should be monitored and a process improvement system should be put in place. The practices described in the Capability Maturity Model [PCCW93] provide an excellent guide for the establishment of a process improvement process.

As one can see, the actual creation of a software development environment based on the principles in this thesis would be a major undertaking. Then again, this should be expected since the underlying premise has been derived in a revolutionary rather than evolutionary manner. A major shift is required in software development to bring it towards a software engineering discipline. There is a wealth of research available to support this revolutionary shift. Adoption of the definition of software engineering provided in this thesis and the establishment of software architectural development as the basis for software development technical activity may provide the focus necessary to bring this large body of research together into a full engineering discipline.

7.0 References

- [BCG83] Balzer, R., Cheatham, T. E., Green, C., "Software Technology in the 1990s: Using a New Paradigm", *Computer*, Nov. 1983, pp. 39-45.
- [Beck94] Beck, Kent, "Patterns Generate Architectures", ECOOP'94.
- [Benington56] Benington, H. D., "Production of Large Computer Programs", *Proc. ONR Symp. Advanced Programming Methods for Digital Computers*, June 1956, pp. 15-27.
- [Best95] Best, Larry, "What is Architectural Software Development".
<http://www.c2.com/ppr/ams.html>, 1995.
- [BMB96] Briand, L. C., Morasca, S., Basili, V. R., "Property-based Software Engineering Measurement", *IEEE Transactions on Software Engineering* Vol. 22, No. 1, Jan. 1996.
- [BMRSS96] Bushmann, et. al., Pattern-Oriented Software Architecture: A System of Patterns, Wiley, 1996.
- [Boehm81] Boehm, B., Software Engineering Economics, Prentice-Hall, 1981.
- [Boehm88] Boehm, Barry, "A Spiral Model of Software Development and Enhancement", *IEEE Computer*, May 1988.
- [Brown] Brown, Kyle, "Using Patterns in Order Management Systems: A Design Patterns Experience Report",
<http://www.ksscary.com/ptrnjml.html>.
- [Card88] Card, David, "The Role of Measurement in Software Engineering", *Proceedings Software Engineering 88, IEE/BCS*, 1988.
- [CN96] Clements, Paul, Northrop, Linda, "Software Architecture: An Executive Overview", CMU/SEI-96-TR-003.

- [Coleman94] Coleman, Derek, et.al., Object-Oriented Development: The Fusion Method, Prentice Hall, 1994.
- [Fowler97] Fowler, Martin, Analysis Patterns: Reusable Object Models, Addison-Wesley, 1997.
- [GS93] Garlan, D. , Shaw, M., “An Introduction to Software Architecture”, Advances in Software Engineering and Knowledge Engineering, Vol. 1., River Edge, NJ: World Scientific Publishing Company, 1993.
- [Humphrey89] Humphrey, Watts, Managing the Software Process, Addison-Wesley, 1989, ISBN 0-201-18095-2.
- [Johnson92] Johnson, Ralph, “Documenting Frameworks using Patterns”, OOPSLA’92.
- [KBAW94] Kazman, R., Bass, L., Abowd, G., Webb, M., “SAAM: A Method for Analyzing the Properties of Software Architecture”, Proceedings of ICSE 16, May 1994, pp. 81-90.
- [Martin90] Martin, J., Information Engineering (a trilogy), Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1990.
- [Martin91] Martin, J., Rapid Application Development, Macmillan Publishing Company, 1991.
- [MJ82] McCracken, D. D., Jackson, M. A., “Life-Cycle Concept Considered Harmful”, ACM Software Engineering Notes, Apr. 1982, pp. 29-32.
- [MM96] Motsching-Pitrik, Renate, Mittermeir, Roland, “Language Features for the Interconnection of Software Components”, Advances in Computers Vol. 43, 1996.
- [PCCW93] Paulk, M. C., Curtis, B., Chrisis, M. B., Weber, C. V. , “The Capability Maturity Model for Software, Version 1.1”, Technical report SEI-93-TR-24, Software Engineering Institute, Carnegie

Mellon University.

- [Pfleeger91] Pfleeger, Shari, Lawrence, Software Engineering: the Production of Quality Software Second Edition, Macmillan Publishing Company, 1991, ISBN 0-02-395115-X.
- [PS94] Peterson, A. S., Stanley, J. L. Jr., "Mapping a Domain Model and Architecture to a Generic Design", Technical Report CMU/SEI-94-TR-8, Software Engineering Institute, Carnegie Mellon University, 1994.
- [PW92] Perry, D. E. , Wolf, A. L. "Foundations for the Study of Software Architecture", Software Engineering Notes, ACM Sig-Soft 17, 4, October 1992, pp. 40-52.
- [RG94] Reingruber, Michael, Gregory, William, The Data Modeling Handbook: A Best-Practice Approach to Building Quality Data Models, John Wiley and Sons, 1994, ISBN 0-471-05290-6.
- [RGI75] Ross, D., Goodenough, J., Irvine, C.A., "Software Engineering; Process, Principles, and Goals", IEEE Transactions of Software Engineering, May 1975.
- [Royce70] Royce, W. W. , "Managing the Development of Large Software Systems: Concepts and Techniques", Proc. Wescon, Aug. 1970. Also available in Proc. ICSE 9, Computer Society Press, 1987.
- [RZ96] Riehle, D., Zullighoven, H., "Understanding and Using Patterns in Software Development", Theory and Practice of Object Systems v. 2, n. 1, 1996.
- [SG96] Shaw, Mary, Garlan, David, Software Architecture: Perspectives on an Emerging Discipline, Prentice Hall, 1996.
- [Sommerville96] Sommerville, Ian, Software Engineering Fifth Edition, Addison-

Wesley, 1996, ISBN 0-201-42765-6.

- [WE96] White, S., Edwards, M., "Domain Engineering: The Challenge, Status, and Trends", Proceedings IEEE Symposium and Workshop on Engineering of Computer-Based Systems, March 1996.
- [WL97] S. A. White and C. Lemus, "The Software Architecture Process", in proceedings of ASME-ETCE 97, The Energy Engineering Symposium of Energy Week' 97, pp. 170-175, Houston TX., Jan. 29 - Feb 2, 1997.
- [Yourdon86] Yourdon, Ed, Structural Design: Fundamentals of a Disciplined Program & Systems Design, Prentice Hall, 1986, ISBN 0138544719.

Appendix A: Requirements for Example Problem

Reprinted from [Coleman94 pages 144,45]

A computer-based system is required to control the dispensing of petrol, to handle customer payment, and to monitor tank levels.

Before a customer can use the self-service pumps, the pump must be enabled by the attendant. When a pump is enabled, the pump motor is started, if it is not already on, with the pump clutch free. When the trigger in the gun is depressed, closing a microswitch, the clutch is engaged and petrol pumped. When it is released, the clutch is freed. There is also a microswitch on the holster in which the gun is kept that prevents petrol being pumped until the gun is taken out. Once the gun is replaced in the holster, the delivery is deemed to be completed and the pump disabled. Further depressions of the trigger in the gun cannot dispense more petrol.. After a short standby period, the pump motor will be turned off unless the pump is reenabled.

A metering device in the petrol line sends a pulse to the system for each 1/100 liter dispensed. Displays on the pump show the amount dispensed and the cost.

There are two kinds of pump. The normal kind allows the user to dispense petrol ad lib. The sophisticated pumps, imported from New Zealand, allow the customer to preset either an amount or a volume of petrol. Petrol will then be pumped up to a maximum of the required quantity.

Transactions are stored until the customer pays. Payment may be either in cash, by credit card, or on account. A customer may request a receipt and will get a token for every 5 pounds spent. Customers sometimes abscond without paying and the operator must annotate the transaction with any available information (e.g., the vehicle's registration). At the end of the day, transactions are archived and may be used for ad hoc inquiries on sales.

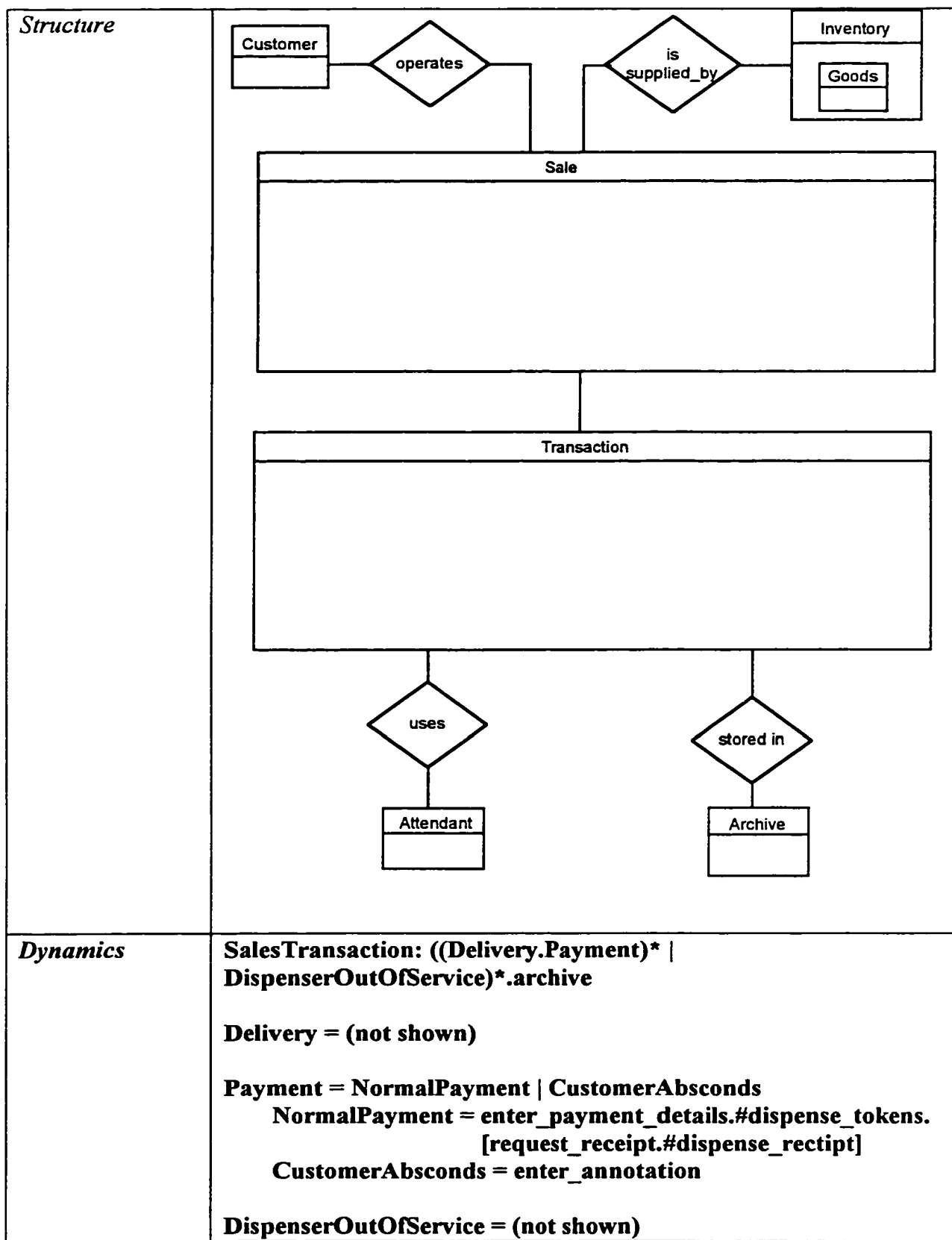
At present, two grades of petrol are dispensed from five pumps on the forecourt. Each pump takes its supply from one of two tanks, one tank for each grade. The tank level must not drop below 4% of the tanks capacity. If this happens, the pumps serviced by the tank cannot be enabled to dispense petrol.

Appendix B: Example Architectural Pattern

The example in this appendix is only a partial pattern definition of an architecture for a sales and transaction system. It is intended to provide sufficient detail to demonstrate the use of architectural patterns in system development as described in this thesis.

Field	Description
<i>Name</i>	Self-Serve Sales Transaction This pattern divides a self service marketing system into two components. The sales component provides the functionality required to service a customer request. The transaction component provides the functionality to process payment.
<i>Pattern Type</i>	Architectural
<i>Also Known As</i>	None
<i>Example</i>	Suppose we are developing a system to sell gas at a gas station. The customer initiates a sale by activating a pump. The system must supply the desired amount of gas and register the purchase. Notice of the sale is sent to the station attendant. The customer then goes to the attendant's window where payment is given for the gas.
<i>Context</i>	A system with independent cooperating components specialized for the purchase of goods supporting self service acquisition of goods and attended payment for goods.
<i>Problem</i>	There are two types of components where one is used to provide an interface for the purchase and dispensing of goods, and the other type of component processes and records the actual purchase. These components are always separated physically and there are usually more than one of each type of component in a given system. The

	<p>purchase component is used by a customer and the processing of the sale component is used by an attendant. The purchasing component must be able to sense the start and finish of the dispensing of goods. Information recorded by the dispensing component must be accessible by the purchasing component.</p>
<i>Solution</i>	<p>Define two component types called "Sale" and "Transaction." The Sales component provides the customer interface and functionality necessary to dispense goods. The Transaction component provides the attendant's interface and the functionality necessary to process the actual purchase of goods.</p>



	<p>Operation Model for each operation in the life cycle model. Model only shown for enter_payment_details.</p> <hr/> <p>Operation: enter_payment_details</p> <p>Description: Receive payment form a customer and dispense promotional tokens, such as value stamps, to the customer.</p> <p>Reads: amount_of_sale</p> <p>Changes: transaction_record</p> <p>Sends: dispense_tokens</p> <p>Assumes: goods have been received by customer</p> <p>Result: Payment is received, the transaction record indicates payment received, and promotional tokens have been dispensed to the customer.</p>
Implementation	<p>Payment Class</p> <p>Sale Class</p> <p>Transaction Class</p>
Example Resolved	
Variants	
Known Uses	
Consequences	
Preconditions	
Companion Patterns	Self Serve Dispenser

See Also	
<i>Constraints</i>	

Appendix C: The RAD Life Cycle

RAD Life Cycle

- 1. Before the Project Begins**
 - 1.1. Select the tools**
 - 1.2. Establish the methodology**
 - 1.3. Select and train the practitioners**
- 2. Initiate the project**
 - 2.1. Obtain commitment of the Executive Owner**
 - 2.2. Establish the IS team**
 - 2.3. Customize the methodology for this system**
 - 2.3.1. Select the appropriate variants of the methodology**
 - 2.3.1.1. Toolset**
 - 2.3.1.2. Timebox**
 - 2.3.1.3. Combination of requirements planning and user design phase**
 - 2.3.1.4. Parallel Development**
 - 2.3.1.5. Reusability**
 - 2.3.1.6. Data Administration**
 - 2.3.1.7. Information Engineering**
 - 2.3.1.8. Object-Oriented Reusability**
- 3. Requirements Planning Phase**
 - 3.1. Preliminaries**
 - 3.1.1. Establish the need for a system**
 - 3.1.2. Determine the scope of the system**
 - 3.1.3. Determine the key user executives**
 - 3.2. Research**
 - 3.2.1. Identify overall objective of the system**
 - 3.2.2. Become familiar with the current system**

- 3.2.3. Explore what changes are needed in the current system
- 3.2.4. Find what relevant information exists in the I-CASE repository
- 3.2.5. Explore the BAA study or similar planning
- 3.2.6. Explore what structures or designs might be reusable
- 3.2.7. Research similar systems that might offer guidance or ideas
- 3.2.8. Create a tentative overview of the new system in the I-CASE repository
- 3.3. Prepare the workshop
 - 3.3.1. Select workshop participants
 - 3.3.2. Prepare the materials
 - 3.3.3. Customize the JRP agenda
 - 3.3.4. Hold the kick-off meeting
- 3.4. Conduct the workshop
 - 3.4.1. Initial review
 - 3.4.2. Determine system functions as a whole
 - 3.4.3. Examine each process
 - 3.4.4. Determine cultural changes that will be caused by the system
 - 3.4.5. Summarize benefits and risks
 - 3.4.6. Determine how to maximize the benefits
 - 3.4.7. Assess the risks
 - 3.4.8. Determine how to minimize the risks
- 3.5. Create the JRP documentation
- 4. User Design Phase
 - 4.1. Prepare for JAD workshop
 - 4.2. Conduct the first design workshop
 - 4.2.1. Create entity-relationship diagrams
 - 4.2.2. Create process decomposition diagrams
 - 4.2.3. Create process dependency diagrams
 - 4.2.4. Create action diagrams
 - 4.2.5. Design screens

- 4.2.6. Identifying prototyping
- 4.2.7. Design reports
- 4.2.8. Determine what data the system uses
- 4.2.9. Determine what processes the system uses
- 4.3. Establish the construction teams
- 4.4. Review and refine the design
- 4.5. Establish the project repository contents
- 4.6. Conduct the second design workshop
 - 4.6.1. Review experience in using the prototypes
 - 4.6.2. Review the end users' questions and suggestions
 - 4.6.3. Discuss enhancements that are necessary
 - 4.6.4. Review the overall design
 - 4.6.5. Finalize design
- 5. Construction Phase
 - 5.1. Construct detail design of the system structure
 - 5.2. Build transactions one at a time with reusable templates
 - 5.3. Perform usability tests
 - 5.4. Perform ongoing integration
 - 5.5. Design and prepare for cutover
 - 5.6. Integration testing
 - 5.7. Create physical design of data base
- 6. Cutover Phase
 - 6.1. Install and adjust the pilot system
 - 6.1.1. Set up the production procedures
 - 6.1.2. Install the production system environment
 - 6.1.3. Perform data conversion
 - 6.1.4. Implement the new system in production
 - 6.1.5. Review the system installation
 - 6.1.6. Expand the pilot to the full system