MANAGING DOMAIN ARCHITECTURE EVOLUTION THROUGH ADAPTIVE USE CASE AND BUSINESS RULE MODELS

BY

RUSSELL R. HURLBUT

Submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science
in the Graduate College of the
Illinois Institute of Technology

Approved ___________________________
Advisor

Chicago, Illinois
May 1998

ACKNOWLEDGEMENT

I thank my advisor, Prof. C. R. Carlson, for his inspiration and guidance during my studies at the Illinois Institute of Technology. Our frequent meetings over the last few years always yielded new insights and provided valuable feedback.

I also owe thanks to several individuals that demonstrated confidence in my software architecture philosophies. They provided me the opportunity to develop actual business domain architectures that conceptually solidified many of the ideas presented in this thesis. These individuals include Clark Riley with R.R. Donneley and Sons, Mike Leatherman with Wallace Computer Services, Paul Bergstrom with Amoco Oil Company, and Ron Martin with William M. Mercer. The experience acquired in architecting such a diversity of systems proved invaluable in formulating underlying principals that are common across many business domains.

Earlier influences in my academic and professional career fostered my intellectual and moral growth. Professor Jim Lauer at Northern Illinois University enriched my interests in rule-based systems and decision support systems that contributed the development of the business rule patterns presented in this thesis. Dr. Carl DeMoon provided me with my first executive management position and the inspiration to develop tenacity, compassion, and a strong moral character. The management and interpersonal skill acquired under his tutelage contributed to a more holistic view of organizational business processes that is reflected in my academic work.

I thank my parents for their nurture and faith in me. I'm sure they would be pleased. I especially appreciated my father's inventive genius and his belief in experience over theory that opened my eyes to inductive reasoning at an early age.

When I began my doctoral program, my business practice was barely a year old. I am especially thankful to my wife Karen for her support during my graduate studies. Without her support, the effort would have been unbearable. As a wife, mother, and business partner, she shouldered a great deal of the burden and encouraged me to persevere. I also thank my two sons, Matt and Rod, who share an interest in my field and make life much more rewarding. Finally, I am thankful to God for having granted me the skills and opportunities that made this possible.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure Page

LIST OF EXAMPLES

Example Page

ABSTRACT

The areas of domain engineering, vertical application frameworks, and business objects have generated considerable interest in industry and the research community during the last few years. In order for systems to be successfully implemented from such application frameworks, there are two major concerns that must be addressed: how to maintain non-interfering applications and how to minimize bias towards any individual application in developing the domain model. As we add new applications, we need to ensure that they are not destructive to each other. We also need to make certain that as we expand the domain model to accommodate new applications, that these future applications are not unnecessarily constrained or complex. Most of the related work to date has focused on defining the architectural structure of application frameworks or on maintaining structural and behavioral consistency. Methodologies have primarily focused on the definition of a domain model rather than its evolution.

This thesis develops a conceptual framework for integrating various techniques to facilitate managing the evolution of a business domain architecture. As part of this conceptual framework, domain normal forms and normalization operations are defined. An adaptive use case model is proposed as an extension to the Unified Modeling Language (UML) specification. A business rule pattern language and meta-model are also developed that describes how parameterized business rules can be integrated with adaptive use cases to mange domain model evolution. All of these models are synthesized into a domain evolution architectural transformation (DEAT) process model. Although considerable work remains, this thesis demonstrates the feasibility of utilizing the prescribed models for guiding performance of operations on a domain model. Through fit assessment and change cost analysis, new applications may be developed from the domain architecture with minimal bias and interference resulting in a more stabile and resilient domain model.

CHAPTER I

INTRODUCTION

1.1. Introduction

As software systems become more complicated, it becomes increasingly difficult for humans to cope with such complexity. Furthermore, programming-in-the-large, which deals with multi-year, multi-million dollar projects, often must deal with situations where significant portions of the system are still under development with none of the original developers. Such development efforts deal with a partitioning of technical knowledge among individuals that address the scope of the project. Within each such partition, the technical leadership may change hands several times, thus further partitioning the technical knowledge over time.

Abstraction, which has become increasingly popular through object-oriented development methodologies, is one way to deal with the complexities of large system development. Domain engineering uses the principle of abstraction to reduce the complexity. It can provide the essential continuity across sub-systems and across time.

Domain engineering, which is concerned with integration of domain oriented components [AR94][1], is a field that has been emerging as the key to any domain-specific reuse strategy [Oops95b]. The generic elements identified during a domain analysis are used during the domain engineering effort to construct design fragments [dCLF93]. These designs can be implemented as domain-specific code or combined into larger constructs such as components and frameworks.

A framework is designed to cover a family of applications or subsystems in a given domain and is typically delivered as a collection of interdependent abstract classes, together with their concrete subclasses [GM95]. Software evolution, along with associated maintenance tools and techniques, is essential to realize the benefits of domain engineering and domain-specific application frameworks. Evolutionary changes to a system can occur for several reasons. The needs of users can change, the underlying real-world domain can change, additional functionality can be added, and the architecture can be improved. The changes may occur during any stage in the lifecycle of the system [Hurs95].

[1] Corresponding to references in the Bibliography

System administrators are often fearful of making changes to a system solely for the purpose of improving the architecture. In order to ameliorate those fears, techniques such as refactoring [Opdy92], propagation patterns [Lieb96], and reflection [Hink94] have been developed to maintain structural and behavioral integrity of the application code when making modifications. Since a domain-specific framework is intended to be frequently reused to create new applications, these perfective improvements may be crucial for continued use of the framework.

By combining domain engineering and application framework development, a domain model is constructed with the intent that it be used by a family of applications. In practice, an initial application is often developed in conjunction with the domain model. Two primary concerns must be addressed in such a scenario:

1. How do we maintain non-interfering applications? In other words, as we add new applications, how do we assure that they are not destructive to each other? If we reduce this to a view problem, it becomes: what extra protocols are needed to safeguard non-interference over the domain of objects?
2. How do we minimize bias towards any individual application in developing the domain model? Building in flexibility that is not needed for an application under developed needs to be balanced against future needs. When future needs dictate a reorganization of the domain model, existing applications may need to be rebuilt to conform to the revised domain model. Therefore, the converse of the question is: how do we minimize the impact on existing applications when evolving the domain model?

Although the severity of these concerns may be exaggerated during simultaneous development of the domain model and initial application, these concerns are still relevant for any domain engineering effort. Some attempts have already been made to integrate various techniques into a common conceptual framework. The Fusion method [Cole94], promoted as a second-generation object oriented methodology, is one such effort. Essentially this approach combines a variety of existing techniques, but doesn't sufficiently explain what to do if a person desires to integrate additional techniques.

The approach taken here attempts to take a more holistic view of the corporate software development environment. It assumes that no methodology provides a complete solution. Instead, the focus is on evolving a domain architecture rather than new system development. This focus will allow the approach presented here to be applied in adjunct to most existing development methodologies. This results in a “synergistic” based approach that synthesizes to achieve new capabilities that are not adequately addressed by isolated techniques. This capability is demonstrated through development of a fit assessment model based upon a novel combination of existing familiar techniques.

Exploiting existing techniques in new ways will enable us to address unforeseen complexity in new business domain architectures. This is critical to the architecture’s success and to the success of the applications that are instantiated from them. Project management does not have an adequate grasp of the technology. Software engineers don’t understand management principles enough to know what they should communicate to managers in order for them to manage the technology. From a technology standpoint, we must bridge this gap and apply management skills to engineering problems. The perceived benefits are of practical use to:

- Technical managers and system architects, who would be aided in determining the ramifications of providing or not providing certain levels of flexibility in the domain specific application framework derived from the domain model.
- Project managers, who would be able to determine resource requirements and risks more accurately when confronted with several simultaneous projects that compete for scarce resources.
- Senior management, who would be able to quantify domain engineering (strategic) and application development (tactical) costs. Such knowledge can aid in initiating organizational changes that optimize relevant business processes.

1.2. Purpose of the Thesis

Software reuse must dominate the development process to more closely emulate well-defined process industries such as electronics manufacturing or steel fabrication. Building reusable assets in the form of components and processes is essential. Furthermore, developing cost accounting practices is needed in order to bridge the cognitive gap that exists between management and software engineers [Venk96]. More rigorous and formal approaches that deliver the information needed by both project managers and software engineers can bridge this gap. However, software development is too varied and complex to be captured by any reasonably small set of problem frames. In order to emulate the practitioners of established branches of engineering, software engineers must practice in a very narrow design space with only small perturbations [Jack94]. There are two potential ways to accomplish this emulation. First, a developer could focus only on a small set of business domains. By becoming sufficiently familiar with this small set, he could address any problem in the set closely and completely. His software engineering skills should chiefly lie in composing the solutions to sub-problems that have been identified and fitted into frameworks of the familiar set. Second, for business sectors deemed suitable for developing a domain specific architecture, the focus should be changed from design principle to design standards. This will probably require industry-wide support for a common base of domain-specific business objects [Wood95]. However, since competitive factors will dictate the separation of commercially sensitive information from common business objects, tools and techniques will always be needed to derive new functionality.

As a domain architecture matures, the need exists for deriving new creative techniques. The difficulty in transforming existing representations into new hybrid ones can be attributed to the lack of a conceptual framework to guide this migration. Although many conceptual frameworks exists that can assist in this process, there is no single approach that can accommodate the unanticipated needs of all domain-specific architectures. It is likely that a long-term research agenda will be required to develop a complete set of formalisms to fully develop the conceptual framework proposed in this thesis (see Figure 1.1). This effort is only taking the first incremental steps in defining such a framework. Extensive empirical evidence in a wide range of domains will be needed to validate the usability of new derived formalisms.

Within the context provided here for framing the problem, a formal statement of the problem would be presented as:

> Given an informal specification in the form of use cases that express the shared intentions of the client and developer, combined with an object-oriented framework derived from a business domain-specific architectural model, provide an approach to identify and utilize software engineering tools and techniques that will i) bridge the cognitive gap between management and engineering practitioners, and ii) support software engineers in applying domain knowledge, design principles, standards, and programming skills to generate an

application and to evolve the framework in such a way that desirable properties are maintained, such as adaptability, usability, non-interference, and compositionality.

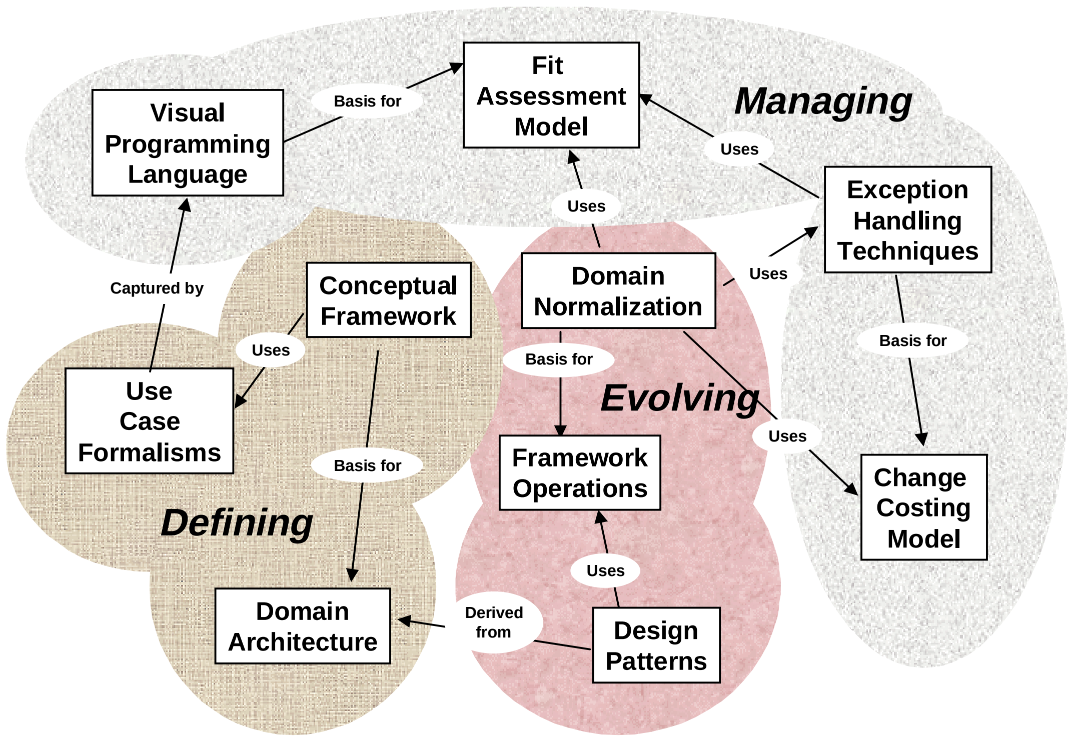


Figure 1.1. Business Domain Architecture Concept Map

Several desirable properties that relate domain-specific frameworks are identified in this problem statement. Adaptability refers to the ability to derive new applications that encroach on the defined domain boundaries, thus requiring new functionality to be added, either in the form of new components or in a new, unanticipated configuration of existing components. Usability, as would be expected, simply refers to the ease in which a framework user or framework developer can interact with the framework. Non-interference deals with the protocols over the domain of objects to assure that derived applications are not destructive to each other. Compositionality is a term used by Casais that is defined as “the specification of components in such a way that their combination enforces global properties automatically” [Casa95].

This problem statement also deals with several entities. *People* playing various roles are expected to apply appropriate acquired *knowledge* to certain software engineering *processes* by taking a set of *artifacts* as inputs and producing another set of *artifacts* as output.

Understanding the relationships between roles is important in identifying the interactions, which in turn enables formulation of the solution tasks. The first interaction is between the *client* and the *developer*. It is the developer's responsibility to satisfy the requirements represented by the use cases. Here, the developer represents the software development organization comprised of both project managers and the software engineers. Once the developer role is decomposed into its component parts, a second interaction between the *project manager* and *software engineer* emerges. It is the responsibility of the software engineer to communicate appropriate information regarding the state of the application development and domain architecture evolution to the project manager. This enables that manager to fulfill his responsibility to the client in his role of developer and to contribute to the evolution of the architecture in his role as project manager. Implicit in this problem statement is the role of the *business domain expert*. The interaction between the software engineer and domain expert primarily involves the evolution of the framework, but is usually driven by the demands of actual application development projects. As such, there is a complex interaction between project managers, software engineers, and domain experts that balances application development and domain architecture evolution.

By examining the roles and interactions, the relevance of the artifacts identified in the problem statement have generally been addressed. *Use cases* facilitate communication between the developer and the client. The *domain-specific framework* is used to generate an *application* and is also the target of additional development effort in order for it to mature. *Software engineering tools* are implicit in the application development processes and the domain architecture evolution.

Three distinct solution tasks can be identified. The first task is to construct an application derived from the domain architecture that satisfies the use cases. The second task is to evolve a domain architecture that can perform operations on artifacts in response to requests. The third task is to construct a system that monitors the development environment to ensure satisfaction of requirements.

In the formal problem statement, the key is to "provide an approach to identify and utilize software engineering tool and techniques". This means that a process is needed that provides the glue to integrate the three identified solution tasks. Applying a conceptual framework to the problem frame and solution tasks enables one to perceive any software engineering technique or methodology from the viewpoint that i) we are bridging the gap between management and engineering disciplines and ii) we are identifying the conceptual relationship between this tool or technique to a common reference, such as “this technique is equivalent to the aspect of the conceptual framework which addresses...” or “this is a pattern that implies...”. Currently, most software engineering approaches really only address design principles. They are too vague with respect to applying their own techniques when stepping outside of the well defined boundaries of their methodology, much less into the complex nature of domain engineering activities. Through this proposed conceptual framework, we can focus on specific aspects within a given methodology to overcome the lack of integration with other approaches, such as *when* to apply a specific pattern, rather than *how* to apply it.

In the existing state-of-the-practice, commercially available software tools and techniques are driving the design process and defining the methodologies. Methodologies have become fluid as software engineering tools frequently change to match features of competitors. Users of these tools are given little guidance in integrating the technique provided within the tools. The “approach” referred to in the formal problem statement represents a “fusion” of what is possible within and among these commercially available tools as well additional techniques that could be incorporated into them. In a very pattern oriented Alexanderian philosophy [Alex79], we are not so much concerned with individual techniques as we are with the patterns of relationships between them. Techniques, which seem like elementary building blocks, keep varying from one domain-specific environment to another – sometimes slightly, sometimes significantly. When we examine the nature of these techniques through a conceptual framework, which grounds our perspective, the techniques themselves dissolve and only the fabric of the relationships and the transformations among them remain.

Domain engineering combines the domain knowledge of business experts with the software engineer’s knowledge of design principles and his programming skills. The approach taken in this thesis makes it possible to combine this knowledge in a well-defined manner based on standard ways of approaching solution tasks. The design process can be broadly interpreted either to imply a rigorous engineering discipline, as in the creation of an office building, or to imply an artistic endeavor as in conceiving an oil painting. The approach to be developed in this thesis will enable us to more closely emulate the features associated with a disciplined engineering process rather than an artistic one.

1.3. Overview of the Chapters

The remainder of this thesis is organized as follows. Chapter two presents a background of domain engineering related fields. Chapter three presents a summary of related works in the area of use case formalisms and business rules. Two tables compare use case formalism focus and formats. A concept diagram ties the surveyed research together. A review of research on the management, evolution, and definition of a domain architecture is also provided. A presentation of the proposed approach for addressing the problem statement concludes the chapter.

Chapter four introduces the domain engineering conceptual framework and the domain architecture normalization operations. A discussion of domain normal forms demonstrates how operations are applied to normalize a domain model. Chapter five introduces the adaptive use case model as an extension to the Unified Modeling Language (UML). A stakeholder/view use case refinement level model is also presented. Chapter six introduces a business rule meta-model and a collection of parameterized business rule patterns. These patterns are presented in a problem/solution format with examples.

Chapter seven brings together the previous three chapters and presents the domain evolution architectural transformation (DEAT) process model. The transformation of use case formalisms from one format to another is described to illustrate how process, structural, policy, and context views of the domain architecture are used to reveal hidden nuances of new system requirements that must be integrated into the domain model. The DEAT process model includes a deployment evaluation sub-model that is comprised of configuration model, a fit assessment model, an exception analysis model, and a change costing model. The chapter concludes with a non-trivial example demonstrating how these transformations may be applied in real system development. Finally, chapter eight concludes the thesis and points out potential future work.

CHAPTER II

BACKGROUND

The focus of this research is on the development of a conceptual framework for integrating various techniques to facilitate managing the evolution of a business domain architecture. Several research areas influence this field of study, including software reuse, domain analysis and modeling, artifact evolution, business objects, and development environments. Additionally, other important concepts, such as frameworks, components, design patterns, business processes, requirements analysis, and reverse engineering provide the glue to bind these areas together in a meaningful way.

2.1. Software Reuse

There is not a lack of software reuse, but a lack of widespread, systematic reuse. Software reuse is characterized by a struggle to formalize in a setting of pragmatic and fast informal solutions. Nevertheless, there are examples of successful reuse such as formal generic mathematics models that solve specific problems across many domains. Formal software reuse through standard, off-the-shelf source-code components were proposed as far back as 1969 [McIl69], but it wasn't until the 1980s that broad interest in reuse surfaced. By then the definition of reuse had expanded to include everything associated with the software development effort. More recently, other peripheral factors such as management, economics, culture and law have been topics of interest to researchers [Cant92], [BB91], [NGT92], [BW88].

Reuse can employ both compositional and generative techniques. Compositional techniques require manual construction of available building blocks. Selection, retrieval, and integration are its key activities. Generative techniques focus on reuse at the specification level and rely on application generators to transform the specification language into code [Gris95].

For any software reuse technique to be effective, it must be easier to reuse the artifacts than it is to develop the software from scratch. In order to reuse any artifact, you must know what it does and where it is. The higher upstream in the development process, the greater the leverage of reuse will be. Therefore, the search for high-level abstraction software artifacts is probably the most crucial research area [Krue92].

According to Yourdan, the typical reuse library contains 200 to 400 components. The state-of-the-practice is for different vendors to be less that fully cooperative. Repositories are often intentionally incompatible through 'dirty tricks' such as retaining locks on everything, undoing changes, using encryption, and reporting misinformation. Lack of documentation for old software means that reverse engineering techniques utilizing expert system technology will probably be necessary to help find useful patterns for reuse [Your93].

The state-of-the-art considers domain analysis and domain engineering as the key to systematic, formal, and effective reuse. Generic specification, designs, and architectures are derived from domain knowledge. These will serve as generic templates for creating components. Process reuse in the form of maturity models and best practices provide the framework and procedures for integration of reuse into the organization. New development paradigms that are based on domain engineering are emerging. Two terms that have been used to refer to new software reuse paradigms are *megaprogramming* and *designware*. *Megaprogramming* is described as a reuse paradigm that integrates library tools, repositories, and techniques into a systematized framework that is domain-specific, reuse-based, and process driven [Prie93]. *Designware* represents reusable application programs provided to the marketplace in the form of fully loaded CASE repositories [Your94].

From a management perspective, as an organization matures in its software development processes, a shift from expense based development to asset based management occurs. Instead of assuming that most of the development cost is derived from the dynamic activities of software development, assets in the form of reusable software and processes accumulate to such a degree that they become the main cost center. With software cost accounting, software development more closely resembles well-defined process industries like electronics manufacturing or steel fabrication [Venk96]. Reuse percent, reuse cost avoidance, and reuse value added metrics can be applied to quantify the reuse effort [PC93].

2.2. Domain Analysis, Modeling, and Engineering

From an analysis point of view, a domain is a well-defined set of characteristics that accurately, narrowly, and completely describes a family of problems. With respect to software development, a domain is a collection of current and future applications that shares this set of common characteristics [dCLF93].

Domain analysis is the key to reusable software in that it stresses the reusability of analysis and design over code [Neig84]. A domain analysis identifies common architectures, reusable components, design alternatives, and domain-oriented terminology. It is expressed in terms of abstract classes and subclasses, protocols, frameworks, constraints, and inference rules, and finally encoded into design schemas. Where appropriate, domain-oriented terminology can be used to create application-oriented requirements language. Adequate domain analysis requires domains with the following properties: well-understood, relatively stable, and relatively high reuse potential to justify analysis cost [Luba91].

A domain model is the product of domain analysis. It provides a problem-oriented architecture for the application domain that reflects the similarities and variations of the members of the domain. An individual target system is created by selecting objects from the domain model to support the provided requirements. The domain model is also used to index into the object repository to ease selection and retrieval. New requirements or variations not present are flagged as unsatisfied. A few of the issues in modeling domains are appropriate levels of abstraction, boundaries of the domain, scope of reusable components in the domain, and identification of the invariant (the *domain kernel*) and variant (bubbles or hot spots) parts of the domain [Goma92], [AR94]. The proper function of the model is to capture how the designers and implementers think about the relationships among the parts of the system, and not necessarily how the relationships are implemented programmatically [BC92].

Domain engineering involves domain-oriented reuse – the combining of software development *for* reuse and software development *with* reuse. A cooperative effort between domain analysts and system analysts is required. Important aspects of the domain engineering are phenomenology, technology of description, and formalization. Phenomenology is the recognizing and capturing the significant elementary phenomena of the domain and the relationships. Technology of description follows the principle of Occahm's razor – describe the parts, such as causality, in as much detail as is necessary, with perfect clarity, but no more. Formalization is choosing and expressing the abstractions and generalization necessary to bring informal reality under sufficient intellectual control for a particular purpose [Luba91].

There is a diversity of opinion as to what artifacts are the products of domain engineering. They range from creation of the domain model to complete application development. Specifications, designs, architectures [Prie93], domain-specific code library [dCLF93], and domain-specific languages and tools [GK96] are three such examples.

2.3. Software Evolution

Software evolution refers to the structural and behavioral modifications to applications. The life span of the average software application is ten years with a large variance – 6.2 standard deviations and a two to thirty year range. Not surprisingly, small-scale systems have a relatively shorter life. The type of application is a consideration in the expected life span. Administrative applications such as personnel and accounting systems live longer than business supporting systems for sales or manufacturing [TT92].

Understanding the reasons for replacing systems can be used to guide software evolution efforts. Satisfying user requirements is one of the causes cited in over half of the replacement cases. Deteriorating maintainability is important, but it is usually combined with other factors such as hardware/architecture change and new technology. However, if software *is* replaced for maintainability, it usually lasts longer. Thus, within the context of domain specific frameworks, an organization must consider the scope, nature, and stability of the domain in order to determine the requirements for the reuse investment [BB91].

With respect to frameworks evolution, it is typically a case of hindsight by which once decides that the class organization can be improved [LH89]. This contributes to the perception that it is often difficult to distinguish between a necessary framework evolution and a design out of control [Ande94]. Nevertheless, there appears to be general agreement that an application framework is a significant advance over previous conceptions of a software toolkit. The patterns of change can be made into principles for framework evolution. Added concepts generate horizontal growth. Additional specialization generates vertical growth [AG90]. However, the current state-of-the-practice is that there is weak tool support for the instantiation and evolution of frameworks [Pree96]. There are two primary reasons for this. First, database systems focus primarily on the data, leaving the update of programs to programmers. Second, capturing the intention of schema changes and assessing their consequences for instances and especially their methods is very difficult. Nevertheless, the ability to propagate schema changes to all the affected components of a system is of great importance [Hurs95].

The state-of-the-art in research holds more promise. Techniques are emerging to reduce the entropy in our legacy base of applications by migrating them into frameworks [FL94], [Berg94]. Algorithms to maintain structural and behavioral consistency during framework evolution make it possible to automate a significant amount of simple framework restructurings [Opdy92], [Hurs95]. The domain-specific kit concept may be used as a base for an evolutionary approach to development by incrementally adding, modifying, and replacing components [GK96].

2.4. Software Architecture

The importance of architecture cannot be minimized. It allows an overall perspective of the systems to be maintained. The lack of an architectural vision is a primary cause of most object-oriented system failures today. Conversely, a strong architectural vision makes it easier to manage complexity, minimize the impact of change, and leverage use of existing components [Stik96].

Software architectures are considered the largest units of reuse [Prie93] and currently come closest to an ideal reuse technology. A “perfected form” of this technology would have small grained domains such as sets and stacks through large-grained domains such as aircraft navigation systems. They offer an advantage over application generators, because rather than creating standalone systems with implicit architectures, software architectures can often be explicitly specialized and integrated with other architectures to create many composite architectures [Krue92].

Conceptually, an architecture consists of models and sub-architectures in the form of heuristics, general principals, and design patterns that provide guidance in developing the models. An architecture should also contain constraints or rules for applying the models and sub-architectures. In order to reduce complexity, some form of cognitive chunking by partitioning various aspects of the architecture should be included. Finally, an implementation arsenal of patterns, frameworks, and tool kits are used to construct these cognitive chunks [Stik96].

The state-of-the-practice is simply that there is insufficient knowledge about the architecture of domain-specific frameworks [Casa95]. The current work done within the Object Management Group for development of a Business Object Facility specification clearly states the case for business objects and an industry-wide architecture [Digr95].

2.5. Object-Oriented Frameworks

The term *framework* has been used as a synonym for terms such as *reference model*, *architecture*, *environment*, and *program template*. This paper will use a definition derived from Ralph Johnson [BJ94]. A framework is the reusable design of a system or a part of a system expressed as a set of abstract classes and the way instances of subclasses derived from those classes collaborate.

The goal of framework design is to build an application from preexisting components. Inevitably, this means that a small number of types of components are reused over and over again. The intention is to write as little code as possible, with the ultimate goal being to build a program by direct manipulation [John93].

Three primary operations can be performed of frameworks: *instantiation*, *refinement*, and *extension* [MN93]. Framework instantiation implements applications from the framework, defining application specific code in the process. Framework refinement, or derived frameworks [Adai95] is an approach to build a very flexible, general framework from which additional frameworks are derived for narrower problem domains. The general framework provides the flexibility, while the derived frameworks provide default behavior and built-in functionality. Framework extension moves in the opposite direction by generalizing an existing framework to simplify the construction of a broader class of applications. Koskimies and Mössenböck proposed the designing of a framework through stepwise generalization [KM95], which parallels the notion embodied by the framework extension operation and adopts Pree's hotspot concepts [Pree94] to focus on identification of flexible aspects of the problem domain.

The state-of-the-practice is that most frameworks that are commercially available have been GUI based. They are generally referred to as application frameworks, which are horizontal in nature and intended to be used across many different domains. Middleware frameworks, such as multimedia and data access have also become prevalent. Support frameworks, which include such functions as file access, distributed computing, and device drivers, are usually buried in the operating system. The domain specific frameworks, which are vertical in nature, such as manufacturing control system, accounting/financial systems, or securities trading, are the most rare. Domain specific frameworks are of primary interest in this research.

2.6. Components

A software component is typically viewed as corresponding to a compilation unit of a conventional programming language or to other user level objects such as files [Shaw94]. They may also be distinguished by their level of granularity. Architectural-level components correspond to subsystems or other independent units. Design patterns may be considered design level components [MDK96]. At the code level, when compared to objects, components tend to be larger, targeted more towards the user than the developer, conforming to industry standard interfaces rather than proprietary, and distributed in binary rather than source code form [Gilb96]. Another key attribute that helps distinguish components from other development artifacts is that they are separable from their original context. Thus, components are usable in other contexts [Kain96].

Components are closely associated with domain engineering. First, domain analysis identifies the vocabulary of the application domain and plays a fundamental role in identifying reusable components [CL93]. Then, domain engineering deals with the integration of these identified domain oriented components [AR94]. Integrating systems from components requires a different methodology and specification technology than building new systems from scratch [MK92]. This usually involves a stable portion and a variable portion that includes architectural variations in the form of optional components or alternative choices among components [Luba91], [BB91], [Come90]. Furthermore, software reuse is more likely if the components perform complicated functions, but appear simple to use [BW88]. In such instances, component-based architectures significantly reduce the surface area of systems, exposing semantics of the business as well as syntax of interfaces [Suth95]. Such large-grained components must maintain a delicate balance between too few and too many parameters. If a component tends towards being a parameter-less black box, it discourages reuse by being to specific and inflexible. If a component provides too many parameters, it discourages reuse by burdening the user with understanding the inside structure in sufficient detail to properly configure the component [LK92].

The state-of-the-practice is to purchase, rather than build, mainstream component libraries that have been developed by third party speculators. Libraries for scientific, operating system support, DBMS, communications, and GUI functions are all available [Wood95]. On the other hand, specific application components rely on expertise within customer organizations rather than with developers, so there is little motivation for liberal reuse/licensing policies [BW88]. Therefore, most everyday programming still defies reduction to component assembly [Jack94]. Reuse efforts limited to the development and utilization of class libraries do not fundamentally affect productivity in the software development process. Lego® snap-together component assembly does not work unless there is some framework to show how the classes fit together [Kors96]. Finding a good division for components in an object-oriented system still seems to be more of a subjective process dependent upon the experience and whims of the designer than an engineering discipline [CL93].

Attempts have been made to compare software components to hardware development – the software-IC analogy [Cox87]. Moore's law of the number of components on a silicon chip would double yearly. Software programming languages have at best realized an arithmetical growth. The reason is that hardware componentization is an iterative process of encapsulating function with each iteration starting at a higher conceptual level. The software industry is constantly creating from first principles [Digr95].

The state-of-the-art shows that research is still needed for generalizing component representation sufficiently to allow reuse over a broad range of target systems and over the whole process of software development and maintenance [LK92]. It will always be technically easier to tune a general component than to generalize a special purpose one [dCLF93]. Furthermore, object-oriented components alone are not very reusable because object-oriented libraries do not scale well. However, some research successes point to a factorization strategy that produces scalable libraries for generating custom object-oriented components [Oops95].

Further work is needed to boost the appearance of libraries of reusable software components for particular application domains [CL93]. Trends include standardization of component structures and interfaces, such as OLE [AAKL93], development of tools to configure framework components, and migration from monolithic to finer grained components [Pree96]. Ultimately products and services will be increasingly supported by software components [Suth95].

2.7. Design Patterns

A design pattern provides guidance for solving a design problem for which no solutions have currently been developed [Schm95]. Typically, a design pattern represents a micro-architecture that applies to a cross-domain design problem. However, some patterns will be specific to a particular application domain [BJ94]. The so called "pattern movement" attempts to codify a reusable base of experience by establishing a common vocabulary for designers through the name assigned to design patterns [GHJV94]. These patterns can then act as building blocks that contribute to overall system architecture design and construction [Coad92]. A set of design patterns forms a pattern language. However, pattern languages are not the same as design catalogs. Whereas design catalogs generate a family of related systems, pattern languages provide a family of individual solution techniques [Cope95].

Design patterns draw their inspiration from Christopher Alexander's pattern approach to building architecture. He created a pattern language to guide the creation of buildings [Alex79] [Alex79a]. These patterns ranged from construction techniques to layouts of entire communities. Alexander describes the essence of his 'quality without a name' in this manner: "It is puzzling to realize that the elements, which seem like elementary building blocks, keep varying, and are different every time that they occur...Look more carefully...to find out what it really is that is repeating there…defined by patterns of relationships among the elements...and finally, the things which seem like elements dissolve, and leave a fabric of relationships behind..."

The state-of-the-practice has shown the emergence of several design pattern catalogs that deliver on the promise of providing the basis for a common design vocabulary. Although three levels of patterns have generally emerged (architectural, design, and programming) [Rieh95], most attention remains on design patterns within the software community.

Capturing this essence that Alexander describes is very difficult. Comprehensive pattern languages for software design require time, effort, and thorough domain analysis to utilize the design patterns effectively [Cope95]. Furthermore, the industry is polluted with attempts to brand any new programming trick a new design pattern. Nonetheless, design patterns have been successfully applied after the first implementation of an architecture to improve its design [GHJV94]. They have also provided value by reducing the effort required learning a class library, since a specific design pattern is typically reused in different places in the library.

The state-of-the-art is to facilitate development of design patterns and frameworks for building business object systems under the premise that domain specific application frameworks will need to have a consistent infrastructure to be widely used within and across industries [Suth95]. Domain-specific pattern languages are expected to become integral parts of a corporate reuse portfolio that enable architectures to be tied to framework classes that instantiate the architecture [Kors96].

Many open issues still need to be addressed. One issue deals with determining whether or not a system of patterns is complete, consistent, or at least provides some minimal range of coverage. Currently, there is no systematic approach to guide the software architect in the usage of patterns [BM95].

2.8. Business Objects

A business object is a representation of something active in the business domain, frequently defined as an aggregation of domain objects. Information that is usually captured about a business object includes its business name, a definition, and its attributes, behavior, relationships, and constraints. A business object may represent, for example, a person, place, or concept. The representation may be in a natural language, a modeling language, or a programming language. The process of business object modeling brings together business process reengineering, workflow, organization modeling, and object-oriented analysis on the basis of an integrated meta-model [Rama95].

The Business Object Management Special Interest Group (BOMSIG) of the Object Management Group (OMG) is addressing the implementation of a distributed-object business system. The Common Business Objects (objects representing business semantics that are common across most businesses) and the Business Object Facility (architecture) are seen as the missing middle layer between the Common Object Request Broker Architecture (CORBA) and custom vertical applications [Suth95]. Two key elements related to this emerging standard are architecture and protocol. The architectural framework must address, among other things, where business rules are put, what happens when a business rule is broken, and what happens when rules or data change. As part of the protocol, issues that must be dealt with include change and dependency propagation, user interface interaction with the business object, how one object locates another cooperative object, common events that drive the system, and error handling [Casa95a].

The intention is for generic business objects with rigorously enforced business semantics to be progressively specialized through common, industry, company, and user business objects. Empowered business users will express their problem in their business language, then rapidly provision their solutions by assembling, specializing, and customizing business application components [Digr95]. An Enterprise Integration Model captures enterprise-specific semantics to allow business application components from multiple business domains to inter-operate.

The key issues involved with business objects are concepts, feasibility, and development. Conceptually, the need exists to identify what business objects to model and how to identify their characteristics. Generally, this means that a business object must be recognizable, relevant to a significant portion of potential customers, and contain sufficient behavior to make it useful. From a feasibility standpoint, it should be non-competitive and relatively stable as a concept. A balance is needed between timeliness, standardization, and abstraction as well as separation of commercially sensitive information that may have become entangled with the basic concept. Finally, determining how to stimulate development and assure quality and usability raise the question of who is best suited to implement business objects [Wood95].

Business object development is in its infancy. The state-of-the-art attempts to integrate business objects into the life cycle of the entire organization from the business planner's perspective. In such a scenario, a repository for business objects (information and rules) becomes the focus instead of reuse of code. Business objects merely become part of a suite of techniques and tools that enable change and learning across all organization functions and levels through a cultural, business, and technical infrastructure [Swan96].

2.9. Requirements Analysis and Engineering

Requirements engineering is concerned with investigating the goals, functions, and constraints of a software system [Zave95]. It can be broken down into four tasks: elicitation, modeling, analysis, and validation [DDB93]. Elicitation of information concerning the problem domain can be carried out though various means, such as interviews, discussions, observations, or perusing available documentation. Modeling attempts to provide a formal representation of the problem. During analysis, contradictions and inconsistencies that relate to individual models from sub-problems of the domain or to the integration of models are identified and resolved. Analysis is also concerned with estimating costs and risks as well as establishing priorities and ranges of satisfaction. The problem statement and results are communicated back to the customer in a comprehensible manner during the validation task. This task helps to assure completeness.

Once these have been identified, the process generally continues with specifying system behavior though complete, consistent, and unambiguous specifications that are well suited for design and implementation activities. The state-of-the-art is to provide automated transformations of requirements into specifications. Although various approaches to transformations have been accomplished in small academic demonstrations for over a decade [Fink88], automated transformations have generally not scaled up well in practical industrial software development.

The ultimate goal of requirements engineering is geared toward reuse: managing the evolution of systems and families of systems. This involves reusing requirements during evolutionary phases for the development of similar systems and for reconstructing requirements. In this context, there are two sets of requirements. The first is the functionality of applications derived from the generic architecture or framework that instantiates the requirement models. The second is the flexibility applicable to the problem domain with respect to reconfigurability [Casa95]. This latter set of functionality pertains to identifying the scope of the framework, and then labeling certain requirements as either necessary, optional, or alternative (mutually exclusive) [Goma92].

2.10. Development Environments

Development environments tailored for domain oriented reuse should address three complementary dimensions: tools and methods, organization and human factors, and productivity programs and measurements [NC89]. Unlike application generators where the customization knowledge is embedded in the macros and interpreters, the design process knowledge is represented explicitly to increase the generality and flexibility of the design environment [BN92]. Development environments are intended to provide several benefits. First, developers should be able to focus more on domain problems rather than on technical ones. Second, the dissemination of expertise captured in the environment significantly reduces the reliance on expert domain specialists. Third, business policies are captured centrally and reused – all derived applications “share the same DNA”. This enables applications to work together more effectively [Tali93].

Since domain experts have different needs than system developers, a combination of techniques should be in the development environment palette [GM95]. A prototypical development environment includes tools that support both the assembly and the running of applications. Browsers, property sheet editors, macro recorders, and application generators are examples of the tools to assist the assembly of applications. Debuggers, interpreters, and monitors are examples of the latter [Gris95].

Several challenges confront the implementation of development environments. One challenge is the balancing of generality with tractability, or efficiency. Generality is needed for a tool to be applicable to a large number of domains. Tractability requires that domain specific tools and languages be developed and applied [Ried90]. As languages become more expressive, tractability becomes worse. Since the combination of these techniques varies for each problem domain, a more holistic approach must be taken. It is quite possible to develop a technically excellent framework and classes that are too complex for the intended users. Application generators, macros, or domain-specific languages cannot be added as an afterthought. The components, framework, and language must all have a compatible design [GK96].

Currently, most domain specific development environments are used for prototyping and proof-of-concept exercises. Research efforts frequently attempt to extend functionality found in related systems such as model management [EK87], [GKS92], repositories [WR92], hypertext [CR92], [CWC94], reverse engineering [Goma93], and knowledge-based systems [MTMS92], [LL92].

Learning aspects of a domain-specific language, especially one in a volatile or immature domain could prove troublesome. Constructing an executable domain over an unstable realm is very difficult and ultimately fruitless. Although some examples of domain specific tools can be found, such as iconic assembly [FNP92], “fill-in-the-blanks” parameterized use cases [Gant90], and non-code models [CEW92], no practical guidance in their use is given. State-of-the-art research is looking at restricting parameterization to a limited number of values and joining related domains into a large reuse infrastructure [DR95].

2.11. Business Process Modeling and Engineering

Business Process Reengineering (BPR) is concerned with redesigning existing operations in such a way as to exploit new technologies and serve customers better. It often involves radical changes to business processes to achieve dramatic improvements in cost, quality, and speed. This concept has become the darling of business management schools. A formal reengineering process usually includes a description of every activity and deliverable involved. Deliverables are usually in the form of business models that focus on the company’s structure and dynamics, and a well-defined process for development.

The state-of-the-practice with respect to domain engineering is the application of an object-oriented perspective to BPR. By focusing on organization goals and underlying decision making processes, applications development planning is facilitated by providing guidelines on how different business processes relate to one another [MMKP91]. Businesses should reorganize along the lines of business processes instead of functional divisions. Object orientation clarifies the company’s products, services, and resources. Use cases provide a natural way to identify its processes [Jaco94]. The state-of-the-art is to combine BPR techniques with cognitive tools, such as mind maps, micro-worlds, and vision models into an object-oriented change and learning methodology [Seng92].

2.12. Reverse Engineering

Reverse engineering is concerned with the understanding, identification, and recovery of software development artifacts [LK92]. The higher the level, such as architectures and domain resources, the more effective the process becomes [LH92]. Reverse engineering, with respect to domain architectures, is closely related to *re-engineering* – the evolutionary process of system development, reuse-supported *forward engineering* – the formalizing of analysis and design with generalizing components and building high abstractions, and *reuse engineering* – producing new software by reusing already existing components [GK92]. Reverse engineer uses the domain model both as a guide and as a recording medium [DR95].

Domain oriented reverse engineering capitalize on the fact that frameworks themselves are implementations, and therefore can provide source code constructs for discovering purpose patterns in the code to reverse engineer. However since programs will typically implement concepts from more that one domain, this advantage fails when dealing with code not related to the domain [DMR94].

In addition to design recovery, reverse engineering has also been used for language translation, documentation, and system restructuring [PBC92]. Techniques to transform procedural programs into object-oriented software [Silv94], second stage reverse engineering to obtain the specification by reconstruction of the functionalities from newly created abstract modules [Cimi92], and object-oriented maintenance-oriented models [HTP89] represent such uses.

More recently, an emphasis on pattern reverse engineering has emerged. "Reverse architecture" is the effort to recover the recurring designs of many software systems and the rationale behind them, as opposed to reverse engineering that tries to recover the design from a single system [Vlis95]. "Pattern mining" can identify four types of patterns in code: containers, high-level design patterns that manage source code dependencies, low-level design patterns, and language specific patterns [Mart95].

2.13. Summary

Domain engineering embraces many different areas and approaches. This chapter has discussed the topic in terms of software reuse and evolution. The organization of applications through frameworks, architectures, patterns, components, and business objects was also presented. Finally, the process of application development was explored through requirements, development environments, and business processes. This summary attempts to weave each of these concepts into a tapestry that both reveals their similarities and their differences.

A business domain is captured through a domain model that defines the architecture of applications that will be created. Although it is possible to have more than one architecture for a business domain, we normally will be dealing with a single architecture in most business domains. Frameworks are a particular way of representing architectures [BJ94], and although there are architectures that can't be expressed as frameworks, they are most frequently represented in this manner. Domain-specific pattern languages then tie the architectures to framework classes that instantiate the architecture [Kors96]. In this way, frameworks capture the common abstractions of the application domain in both structure and mechanisms [Cope95].

Class library components are discrete, context and application independent solutions to a large range of problems. However, they can be utilized by a framework to specialize behavior for a particular application instance or they can help frameworks interlock with other frameworks through shared abstract classes [JF88].

The ultimate goal of a domain engineering environment is to develop a sufficiently robust domain model such that new applications can be instantiated from the framework by simply configuring pre-existing components. These components would either be generic business objects that have become available industry-wide or have been crafted for previous applications. If requirements analysis has adequately anticipated future needs, flexibility will have already been built into these components through parameterized behavior. Development environments that allow configuration of the components and specification of parameters will come as the result of a maturing domain specific suite of tools and language. In this way, systems will realize more efficient reuse of existing artifacts and provide for a more graceful evolution of the business domain as may be dictated by organizational changes through business process engineering.

Finally, as certain domains mature, the need to bridge differences in representations will become a more prominent issue. Reverse engineering techniques can play a role in identifying common design patterns that may facilitate such an effort.

The purpose of this background chapter has been to present the diverse issues that confront the domain engineering effort. Without an understanding of the immense landscape that this field of research attempts to embrace, it is all too easy to become myopic with respect to the scope of available tools, techniques, and resources available. The conceptual framework and domain engineering techniques proposed in this thesis rely on the synthesis of a myriad of techniques to reveal insights to particular problems that will be confronted in business problem domains. Since these problems cannot be adequately anticipated in advance, a practical guide to integrating and using previously unrelated techniques becomes indispensable. It has been the intention of this chapter to lay the groundwork to make such conceptual connections. The next chapter continues this process by exploring similar efforts that manage this complexity though use case formalisms, business rule models, cognitive models, and frameworks.

CHAPTER III

RELATED WORK

The purpose of this chapter is to describe research relevant to this thesis. Use case formalisms may be classified in many ways. The approaches surveyed present a myriad of classification dimensions. Use case formalisms and related techniques are viewed from a broader perspective of defining, evolving, and managing domain architecture artifacts. Figure 2.1 provides a concept diagram that embraces many of the ideas presented in the approaches surveyed. There are two dimensions that have been chosen that provide sufficient common ground to allow a meaningful comparison to be made. The first one is the primary focus of the approach with respect to a use case model's static, dynamic, and policy properties. The process of use case modeling is also considered in this dimension. Table 3.1 summarizes this focus comparison. The second dimension compares representation formats. These formats may be textual, graphical, or dynamic in nature. Table 3.2 provides a comparison of use case formalism formats. These tables merely denote the presence of some measure of meaningful coverage by each approach. The intent is to provide an indication of overlap of concepts. Relative merits are not indicated in the table. The representative research is grouped into the following categories:

1. Use case formalism focus
2. Use case formalism formats
3. Defining the domain architecture
4. Evolving the domain architecture
5. Managing the domain engineering process
6. Conceptual frameworks and methodologies

3.1. Use Case Formalism Focus

Use case focus areas have been categorized into four general properties in Table 3.1: static, dynamic, policy, and process. Although each approach will demonstrate characteristics of each of these four general areas, the basis for comparison is derived from the approach bias for the evolution of use case models either explicitly or implicitly expressed by the authors.

3.1.1. Static Properties

The static properties are concerned with how use case model elements relate to each other. Inheritance and aggregation are familiar object-oriented concepts. Dependency and refinement are two concepts that are well documented in the Unified Modeling Language (UML) specification [UML97]. Data is concerned with the structure of information that is exchanged between actors and the system. Context is concerned with how use case model elements relate to each other.

Jacobson's recent collaboration with Griss represents the major reference to use case modeling using UML notation [JGJ97]. In particular, their enumeration of various mechanisms for introducing variability and discussion for the construction and reuse of use case components have provided significant advances in the maturity of use case modeling. With its close coupling to the emerging UML standard, their work also provides valuable insight as to how and why the UML use case semantics are presented as they appear in the current Version 1.1 specification. However, along with being a major shaper comes the burden of hanging on to conceptual constructs that have persisted in confusing use case modelers. In particular, this is a reference to the «uses» and «extends» stereotypes. Conceptually, they made much greater sense when use cases were highly informal narratives. However, with the strict semantics of UML semantics, many believe that these stereotypes have outlived their usefulness. For example, Cockburn establishes a dependency relationship for the «uses» stereotype by making it a single line item in a scenario [Cock97]. This could just as easily appropriately be called «invokes» [HSFG97].

Two other meta-models for use cases were presented – the OML use case model [HSFG97] and the Regnell use case model [RD97]. The former abstracts an essential use case model element, which are similar to the archetype scenarios of the adaptive use case UML extensions presented in this thesis and the key scenarios from Coad's strategies and patterns [CNM95]. The latter defines and episodes and actions, which again resembles the adaptive use case meta-model's decomposition of use cases into action segments and delta scenarios. Change cases [EDF96] introduce a slightly different construct that deals with versioning of use cases.

3.1.2. Dynamic Properties

Most of the dynamic properties selected also correspond directly to UML semantics. State, event, and action are explicitly defined model elements that relate to state machines. Contracts/patterns and roles are derived concepts that relate to the UML collaboration model. The remaining two properties are not well addressed in the UML specification. Transactions are larger aggregations of interactions that may provide a basis for decomposition of a use case. Agents add mentalistic concepts such as beliefs and goals, which are more typically associated with workflow and business processes.

Several papers discussed business processes and workflow issues. In particular, the agent based use cases [KMJ97] demonstrated Integration Definition for Function Modeling (IDEF) equivalencies with use cases, allowing integration with control flow, agents, and plans. The UML extensions for business process modeling could have benefited from contributions in the workflow community rather than stretching the use case and analysis object types from Objectory [JBJE95], [JEJ94]. For example, the ActionWorkflow process models are the product of a long history of workflow automation research by Winograd and Flores [MWFF92]. Their speech act model [WF87] provides a more robust representation of use case outcomes that the success/failure dichotomy of Cockburn's use case model. The state of the dialog will tend to modify goals and place definitions of success and failure within a context.

Several other examples of business process treatments include task scripts [HSFG97], the addition of scenario steps that occur outside system boundaries in order to incorporate workflow [KPW97], abstract business processes [GE97], event driven workflows [FCS97], and the notion of persistent workflows [Beed97]. It could also be argued that the reification of use cases [Jans95] represents a form of workflow automation similar to Beedle's persistence of workflow objects.

Several approaches are role based. Chafi’s object role stereotypes provide an alternative to those presented in the Objectory and Business Process Extensions to the UML [Chaf96]. An approach dealing with role composition can be found in [VN96]. Holland's work on component contracts provides a formalized basis for interfaces that have been adopted in many other approaches [Holl92].

State machines played prominent parts in many approaches. Two approaches address the relationship between use cases and state machines from opposite directions. One elicits a state machine from use cases [ML97], while the other generates formalized use cases based on state charts [Glin95].

3.1.3. Policy Properties

Policy properties relate to business rules, which are subdivided into three types. First, behavioral rules deal with invariants, pre- and post-conditions. Exception handling makes a distinction between alternative courses of action and exceptional courses of action, including failures. This property is a focus of [SSP95]. Composition relates to functionality that is included rather than how it is enforced. Use case components [JGJ97], adaptive frames [Bass97], and structure blocks [AB95] represent various mechanisms that guide declarative composition of use case model elements. Commitment-based software development relies on a more interactive architecting approach to meet requirements [MTMS92].

Although definition pre- and post-conditions is relatively common, most approaches only give passing references to business rules. Two approaches apply rules directly to application objects [ILOG97], [HSFG97]. Competing rule meta-models can be found in [Herb95] and [HH95]. Silva and Ross present two classification schemes for business rules that decompose rules into atomic constituent parts [Silv95], [Ross97]. Nassiff takes a slightly different approach to classification dealing with such aspects as authority and importance [Nass91].

3.1.4. Process Properties

The final general category, process, is concerned with the development process and participants. As stated at the beginning of this section, evolution, is the key basis for this comparison. Several aspects of use case evolution have been explored. Three noteworthy examples are maintaining a history of use case evolution [ADP95], metrics for measuring the impact of change [EDF96], and functionality distribution [RD97]. Another reference to metrics concerned use case points [HM95]. Process standards are addressed in [Zeie95]. Negotiation for scope and timing of functionality is closely related to evolutionary concerns. Commitment-based software development concerns the process of negotiation in which implementation decisions are changed to meet new specifications [MTMS92]. Use case components assume dependency on requirements that closely match available components. This often results in either changing application requirements in small ways or negotiating more substantial changes [JGJ97]. Change cases involve a process that considers budgetary constraints when defining change cutoffs that tend to limit upstream changes [EDF96]. The fit assessment model presented in this thesis follows similar rationale with respect to changes.

Parameterization is closely related to evolution, since its intent is to provide future variability. Parameterization was the most frequent approach for introducing variability into use case formalisms.

Pattern based variability [Fowl96], [GHJV94], adaptive frame based parameterization [Bass97], [JGJ97], and instance parameters [Beri97], [AB95], [RD97], [DW97] are representative examples of this property.

The *Users* category addresses those approaches that elevate the importance of the user relative to the modeler. Scenario trees [HSGK94] address all scenarios for a particular user. Synthesized usage models take a multiple-role, single-actor view [RKW95]. Dialog maps represent a possible user interface [Wieg97]. Service usage models describe the dynamic behavior of the system services from the user's perspective [KS94]. Baseline requirements documents define user requests as operations on these documents [BuH89]. Collins' iterative use case prototyping approach cites use case refinements as user requested changes [Coll95].

The *Tools* category simply refers to approached that describe automated tool support for their approach. Automated tool support to handle transformations was discussed in [KHT95], [JGJ97], [Beri97], [KS94], [DDB93], [JB93], and [AP92]. Verification through automated tool support was another area that was well covered in [HM95], [Mend95], and [DWMW96].

3.2. Use Case Formalism Formats

Table 3.2 identifies several types of representations for use cases. Although each of the approaches can be fitted into one or more of these categories, the rich variety of representational schemes merits special attention. A basic argument to support the plethora of model elements and diagramming techniques can be found in Horn's information mapping techniques [Horn89]. First, no two domains are exactly alike. This notion is supported by Jackson's problem frames [Jack94]. Second, different personalities playing the various stakeholder roles can relate better to certain types of representation. Third, the maturity of a domain architecture influences the applicability or certain representations. Having a toolkit that consists of a wide selection of such techniques facilitates understandability, reuse, and accuracy. However with such an array of representational devices, it becomes imperative to have a single underlying meta-model that supports the transformation from one representation scheme to another.

3.2.1. Textual Formats

Five categories of textual format are defined in the table. Purely textual formats are unstructured text narratives or semi-structured scripts. Some approaches dealt with a restricted English format [Cock97], [Grah95]. Structured descriptions refer to a form that is used as a template. Those papers that emphasized this aspect were [Harw97], [Cock97], and [ABS95]. Tabular formats are similar to a database table, such as for state-event matrices. The use case summary matrix [ABS95] and key event dictionary [WF97] are two such examples. Formal language expressions are regular expressions in a well-defined syntax. Three papers developed such regular expressions conforming to strict formal syntax. Beringer's regular expressions are used for representing object life cycles [Beri97]. Path expressions describe the interactions of a group of cooperating objects [AC94]. Formal expressions are also used to describe collaboration refinement [DW97]. The object constraint language that was incorporated into the UML specification would be another example of a formal language expression.

3.2.2. Graphical Formats

Graphical formats include structure, state, interaction, and implementation diagrams similar to those defined in the UML notation guide. One additional type of diagram, the rule diagram, is included for graphical representation of rules. Graphical representation of rules can be found in [Silv95] and [Ross97]. Several new modeling constructs were introduced. These included the use case and actor components [JGJ97], use case summary goal groups [Cock97], and two types of model elements referred to as episodes [KPW97], [RAB96]. All of these modeling constructs are intended to enhance reusability. Many different diagramming techniques representing new types of use case models were also described. These included directed use case graphs [KPW97], scenario trees [HSGK94], message sequence charts [AB95], service usage models [KS94], dialog maps [Wieg97], use case maps [Buhr97], context diagrams [ABS95], functional area models [ABS95], and use case dependency diagrams [ABS95].

3.2.3. Dynamic Formats

Dynamic formats imply automated tool support for the assembly, visualization, and navigation of use case model elements. Automated tool support for dynamic formats is immature, so these categories are mostly just propositions. Animation visualization is discussed in [ADP95] and [Thom95]. Hyperlinks navigation is discussed in [ADP95], [Zeie95], and [Horn89]. A graphical browser of scripts and models is described in [BB95]. Use case driven dynamic assembly is covered in [Buhr97].

Two other types of dynamic formats that may not require automated support are role-playing, such are with CRC cards, and storyboards. A storyboard differs from visualization in that there is no animation involved. Storyboards relate more closely to a slide show. Zorman makes use of storyboards [Zorm95].

Wieger's use case dialog maps may also be considered a form of storyboard [Wieg97]. Two role-playing examples are component partitioning validation through Class-Responsibility-Collaboration (CRC) cards [DWMW96] and the “act it out” strategy [CNM95].

3.3. Defining the Domain Architecture

The particular problems of requirements engineering as they relate to the evolution of a domain model can not be adequately covered by a discrete set of approaches. Each domain has its own unique set of issues and problems to overcome that can render a generalized approach less than satisfactory. At best, we can hope for a suite of tools and techniques that can be selected for any particular problem domain. Still, the need to identify proper boundaries and aspects [AR94], [JB93] is a first step. Comer's generic architecture approach [Come90] and the RECAST Generic Application Framework [NGT92] provide process guidelines in performing informal fit assessments. The former includes knowledge of previous systems as an important ingredient; the latter combines the requirement collection process into the selected framework in what could be equated to a “best practices” template. These approaches assume a complete set of frameworks have been provided. However, new problem domains may require new architectures. Unresolved issues from these approaches are the development of new generic architectures, finer refinement of generic architecture in the initial stages of domain engineering, and the consequences or alternatives available if the initial selection of an architecture is deemed inappropriate. Integrating the Jackson problem frames [Jack94] and the structured refinement requests approach [BuH89] with generic architectures could potentially yield a technique to address these issues.

Numerous techniques exist for managing the reuse of requirements. Each of them can have some influence on the evolving domain model that embodies the requirements. However, four of these are of particular interest because of their novelty and how each relates more effectively to different stages in the domain model's evolution. The Comet system’s [MTMS92] approach of interactive design evolution through context specific guidance to support design decisions has special appeal to an immature domain model. Aries [JB93] utilizes influence analysis driven simulations that specify appropriate levels of abstraction and that validate scenarios and anti-scenarios, which are more appropriate in a maturing domain model. The exemplar bidding process [LS94] relies on a rather mature domain where new functionality is more the exception than the rule. Finally, ALBERT [DDB93] relies on black box problem specification with recursive refinement through identification of behavior preserving subsystems, which are most appropriate in a highly stable domain. Each of these, as well as several others, offers a technique that can be a valuable addition to the domain engineer's arsenal of tools. The challenge is to either integrate them into one environment or to provide some mechanism to migrate from one technique to another. Because a single, all-purpose environment is inappropriate for the same reasons that no single requirement elicitation approach can be used in all cases, some form of technique migration is more practical. Information preserving properties that would allow one approach to mutate into another will likely require extensions to the techniques.

A synthesis of the Comer adaptation types [Come90] with Gomaa's domain requirements categories [Goma92], especially those flagged as unsatisfied, could provide a framework for a tactical evolution strategy to address the next iteration of domain model development. This synthesis could also integrate the SCRUM discovery process of dividing requirements into packets [Schw95]. Thus, sets of commonality requirements, adaptation requirements, and complementary requirements can be used to validate this next domain model iteration.

Several related research efforts provide domain specific architecture support. The Software Technology for Adaptable, Reliable Systems (STARS) attempts to remove feature glut in its approach [Lips96]. The Feature-Oriented Domain Analysis Methodology (FODA) maps a domain model and architecture to a generic design [PS94]. The ADAGE project prescribes an integrative model that relies on software system generators for the target domain [BMCT95]. This is similar to the ELDC approach [Goma93], but contrasts the generative approach found in DRACO [Neig84] and GenVoca [Sing96].

The requirements elicitation process itself can go through an evolutionary process. Practitioner methodology [RWM+92] research illustrated that the problems of automatically organizing requirement questionnaires and maintaining a stable domain vocabulary are key aspects to a domain engineering environment. Fuzzy logic and pattern matching may be helpful in generating structured questionnaires, but this seems more appropriate in a maturing domain model than in earlier stages. The concept maps [UM91], [BuH89] appear to be an effective early stage technique that could easily integrate into the conceptual modeling environment of ALBERT [DDB93], which includes software components, manual procedures, real-world entities, and specific hardware devices as part of the conceptual domain. Synthesizing the language formalism discussed in [Fall93] and [SC95] with other techniques, such as formatted requirement [Fink88], the Practitioner questionnaires, or iconic metaphors [HC91] appears to offer some intriguing possibilities for evolving a formalized domain-specific language consisting of grammars and graphical symbols. Guidelines for providing automated assistance in developing a multi-dimensional language could be a valuable contribution to the fields of requirements and domain engineering. One possible approach is to combine presentation and knowledge structure into one language similar to SGML [Jone91]. Such an approach offers an advantage over a language definition such as BNF. This could be used for a tool to convert requirements into user friendly presentation and also into internal computer representations. This would be similar to how Microsoft Frontpage or Netscape Gold works for HTML editing or for database query tools that generate SQL from point and click interfaces.

3.4. Evolving the Domain Architecture

Although a significant portion of research concerning software evolution involves frameworks, every other aspect of the system development life cycle has also been considered. Such research includes the evolution of requirements [MTMS92], [JB93], specifications [EvBD92], components [Wood95], classes [Casa90], [GTC+90], and objects [Auge91], [HJS92], [Pern90]. Even the development process itself has been studied for its evolutionary processes, such as work structures [KCM91], roles [CC93], aspects of the development environment [PT90], and the organization as a whole [Seng92]. Re-engineering can also be considered an evolutionary process of system development [LK92].

There has been a considerable amount of cross-fertilization among the adaptive software [LBS90], [FL94], [Silv94], [Hurs95], [Lieb96], refactoring [Opdy92], [JO93], [OJ93], [Moor96], [RB95] and software communities research efforts [Casa95], [DR95], [JF88], [NGT92]. For example, Casais references the Johnson/Foote design reuse strategies [JF88] and the Law of Demeter [LHR88]. He relies on the Law of Demeter for some of his evolution/reorganization guidelines, although he states that it can only be partly automated [Casa90]. The Evolutionary Domain Lifecycle [Goma92], [Goma93] differs significantly from the others by demonstrating a bias towards a compositional strategy. By synthesizing the Murphy and Notkin framework refinement and extension operations [MN93] with the opportunistic parameterization techniques of the class dictionary graphs [FL94], we can integrate the concepts of adaptive refinement (implicit parameterization) and adaptive customization (explicit parameterization).

This combination can potentially rearrange the relationships among the concepts of framework refinement, framework instantiation, and framework completion.

Framework refinement and extension operations can be supported by automated refactoring operations. Opdyke cautioned that when refactoring to generalize, analogy-based systems are only useful in well-defined domain models [Opdy92]. This argument dovetails nicely with the legal analogy, discussed by Gibbs, *et al.*, which is based on the reusability of past experiences and continuously evolving legal precedence in the form of past ideas, arguments, and cases [GTC+90]. A domain architecture begins to mature after it has been used to generate several applications. New frameworks may be derived from automatically refactoring these existing applications. These new frameworks not only maintain behavioral consistency; they can also substantially reduce the parameterization effort through the clustering of identical sets of parameter values into meta-parameters. Meta-parameterization essentially represents the “legal precedence” established by recognizing an architectural pattern in the existing applications. Gibbs made a distinction between component development and application development. This thesis presumes that framework development represents a third type. Framework refinement and extension should have equal significance for domain engineering in order to exploit this analogy-based meta-parameterization capability.

Murphy and Notkin made a comparison between framework layering and framework operations, particularly framework refinement. They question how to guarantee that a refined framework is substitutable in both its structural and behavioral relationships when the application already has several other instantiated frameworks [MN93]. Combining refactoring with formal use case techniques may be able to overcome these concerns.

3.5. Managing the Domain Engineering Process

Johnson and Russo strike at the heart of the issue with respect to domain engineering strategies [JR91]. They discuss organization factors including management support and justification of a separate team when framework users are outside the organization. However, they do not address how to minimize the "semantic earthquakes" [DR95] of new abstractions that may force a reworking of the framework structure which have been reported to occur [BE93]. Furthermore, they do not describe how interactions with domain experts affect the evolution of the framework as it grows to provide more complete coverage of the analyzed domain. Prioritization of sub-domains is not considered beyond interest for pilot projects. They cite economic reasons for parallel development of an initial application with the framework, but do not consider the impact of changes to the framework on existing applications. Johnson does address this last point in subsequent refactoring research [JO93].

Taligent's derived frameworks concept [Tali93] creates narrower problem domains from a very flexible, general framework. When contrasted with adaptive frameworks [Lieb96], the derived framework paradigm is inherently more compositional, since these narrower frameworks plug into a larger general framework that maintains the thread of control. Adaptive frameworks are more monolithic in their nature of providing constrained combinations of components that define the refinement.

When disparate teams are developing applications instantiated from the same framework, expediency may dictate that bug fixes or enhancements be made directly to derived classes in a "programming-by-difference" scenario. The same bug may have been fixed several times in several different ways due to meeting project management deadlines. How the reconciliation of multiple branches is addressed depends on whether a compositional derived framework strategy or a monolithic adaptive framework strategy is employed. The compositional derived framework strategy can provide a tactical advantage for the immediate project timeline, but suffers strategically through inconsistency in architectural style [PA95]. Deriving a new framework refinement will take more initial effort, but maintains a higher maturity in domain evolution. The relative weight of these tradeoffs will vary depending on how far the domain architecture has evolved. A quantitative analysis mechanism could guide the appropriate strategy from a project management perspective by considering both immediate application development and domain architecture evolution needs.

Taligent's notion of "derived frameworks" also promotes default behaviors through framework completion operations [Adai95], [Tali93], [Tali94]. However, the issue of covariance [MN93] is not addressed: if a more specific framework than the one prescribed is attached, could (or should) the calling framework depend on the additional behavior being present? This issue is related to Basset's construction-time binding issues [Bass96]. Basset's pruning of properties by frames runs contrary to how most object-oriented languages deal with class hierarchies, but it scales up well when these frames are implemented by larger abstraction, such as framework components. Combining his archetypes and deltas technique with framework components provides intriguing possibilities for constraining the concrete subclasses available at any given level or context in a framework.

The Koskimies and Mössenböck stepwise generalization approach moves in the opposite direction from derived frameworks [KM95]. Since problem generalization is accomplished through specification of a representative application, their approach is particularly well suited for parallel development of an application and the domain architecture.

Very few attempts have been made to formalize documentation for domain architectures. The use of motifs [LK94] and exemplars [GM95] attempt to provide more structure to documentation, but fall short of prescribing a formalized approach. One reason for this appears to be a widely held belief that framework documentation should be kept informal [CI92], [John92], opting towards more of a cookbook usage approach. These approaches do not directly address the need to relate documentation to different audiences: the domain expert, the framework developer, and the framework user. The technology books and product books approach [ASP93] at least discriminates between the developer and user. Unfortunately the product book, which captures specific versions of relevant technology books to match an individual system instance, is mostly derived after the fact.

The basic templates and forms approach [GS88] also attempts to provide a usable structure to documentation without any formalism, relying on informational links as an important component. This approach could be combined with product books to create a more robust process to dynamically capture and access documentation during the development process. However, this still leaves the presentation of complex sets of information unresolved. Regardless of the level of formalism, any documentation strategy should rely on the ability to chunk information into digestible pieces. Design patterns have been successfully used in this manner by embracing a sometimes-complex set of interactions with a single representation. The Horn “Information Mapping” approach [Horn89] could provide a conceptual framework for tying together a multitude of representations that may be needed to capture the essence of the unique needs of a specific domain. If an environment could dynamically capture documentation and then provide immediate access with appropriate representation through chunking, then the addition of use case formalisms could foster a rigorous technique for assessing anticipated enhancements and modifications to a domain architecture at any level of its evolution.

One of the most revealing aspects of case studies of domain specific frameworks was a general consensus that object-oriented analysis and design methodologies alone are not sufficient to address the specific needs of framework construction [AG96], [Casa95], [BvA94]. All three of these case studies subscribe to enhancing OOAD with a layered framework approach. Betlem and van Aggele further recommend composition filters that integrate database features with object states [ABV92]. These recommendations provide a great incentive to explore combinations of techniques though synthesis of their most appealing features. Casais's call for enhanced and formalized use cases [Casa95] is another example of the recognized need to combine techniques. Many other examples have already been suggested in this section. However, without a conceptual framework to manage and assess the combination of features, the realized gain may not be worth the effort.

One interesting observation from most of the case studies was the unbridled enthusiasm for their success, such as suggesting that the domain-specific kit concept is an evolutionary step beyond frameworks [GK96] or demonstrating a framework's conformance to international standards [MMT94]. These case studies contrasted with the Shindyalow paper [SCCB95], which provided a detailed and frank discussion of framework development problems. Often, more can be learned from failures than from successes. Other efforts to identify pitfalls [Webs95] or anti-patterns [Koen94] further suggest that the concept of anti-scenarios, which represent inappropriate use case dialogs, may be worth pursuing as a technique.

Another aspect of the case studies was the coincidental implementation of functionality that may be usable at the framework development level, such as the truth maintenance [WS93] and constraint solving toolkit [HHMV93]. Sometimes, the inspiration for connecting unrelated techniques is serendipitous. The KASE system mechanism to learn algorithmic descriptions of the design process [BN92], the DEROS application design assistant [CWC94], the ROPE-ADE evaluation and revision assistants [Gant90], or the model/agent pairs [GKS92] represent similar types of approaches that have already been successfully demonstrated. The search for potential techniques to integrate into the domain engineering environment should not be restricted by conventional thinking.

An "application generator" generator [BuH89] and programming languages for writing domain specific software generators [Sing96], [WL97], [BMB97] focus of transformations that permit domain models to become executable applications. These concepts are on the periphery of this domain engineering research, but since they provide a context for implementing techniques that are central to this thesis, they are included for completeness.

3.6. Conceptual Frameworks and Methodologies

The conceptual frameworks presented here run the gamut from being very concrete and specific where applicability is obvious, as in the transforms business strategies transformation framework [HBPP95], to a highly abstract information model [Horn89]. It should not be surprising that the information mapping conceptual framework, which is so unrelated to domain engineering, should in fact be a better framework from which to base the conceptual framework developed in this thesis. As presented in the business object discussion in the previous background chapter of this paper, the state-of-the-art is to address the learning needs of the organization [Swan96].

Berre presents an object-oriented framework for systems integration consisting of a distributed heterogeneous object management system that provides transparent objects. Transparent objects can represent information and functionality in underlying connected systems such as databases as encapsulated objects [Berr92]. Moffett and Sloman construct a conceptual framework that focuses on policies [MS91]. They describe a need for an independent manager to negotiate, establish, query, and enforce policies that apply to a defined general set of situations. There are two main policy modes: motivate actions and give authority (power) to execute actions.

3.7. Proposed Approach for Addressing the Research Problem Statement

This chapter has analyzed various tools and technique with respect the their applicability to the definition, evolution, and management of a domain architecture. Comparisons of research efforts were made to identify possibilities for integrating the strengths of various approaches as a means of addressing inherent weaknesses. This analysis was intended to provide a basis for the development of a conceptual framework for managing the evolution of a business domain architecture. This framework is developed in chapter four. Understanding how a large variety of techniques can be combined reveal patterns that form the basis for this conceptual framework. In particular, sets of commonality requirements, adaptation requirements, and complementary requirements derived from [Come90], [Jack94], [Goma92], [MN93], [FL94], and [Schw95] are synthesized and then extended into a set of domain normalization operations for evolving the domain architecture.

Once the foundation for evolving a domain architecture has been established, then several key mechanisms must be put into place to facilitate the management process. This review has pointed out the weaknesses in the «uses» and «extends» stereotypes for use cases [JGJ97]. These limitations are overcome with a more flexible mechanism through the adaptive use case model presented in chapter five. The Basset frame technology [Bass97] is adopted and modified to provide the variability mechanism in a manner that is specifically tailored for describing use cases. Additionally, the speech act model [WF87] is integrated into the adaptive use case model to provide a more robust representation of use case than that found in [Cock97]. This represents a synthesis of the best features from object-oriented development methodologies and workflow process approaches. Another contribution of the adaptive use case model is the development of new graphical and textual representations for use cases. As pointed out in the discussion of Horn's information mapping techniques [Horn89], no two domains are exactly alike. Thus, these additional representation devices provide versatile tools for visualizing and understanding the domain model that may be added to the arsenal of domain model tools and techniques.

The second key mechanism that is spawned from the domain engineering conceptual framework is the business rule model and related parameterized business rule patterns. In chapter six, a business rule model is presented that integrates aspects from the Herbst and GUIDE rule meta-models and extends them to be compatible with the UML meta-models [UML97]. Moreover, a pattern language of parameterized business rules subsumes the Silva and Ross classification schemes [Silv95], [Ross97]. These parameterized rules provide a powerful specification tool that also incorporates the rule authority [Nass91], language formalism [Fall93], [SC95] and formatted requirements [Fink88] approaches described in this chapter.

Chapter seven brings together the adaptive use case and business rule models from chapters five and six into an integrated process model for managing domain architecture evolution (the DEAT process model). This process model demonstrates the application of the domain normalization operations from chapter four through its configuration and fit assessment component models. These sub-models provide a basis for measuring the previously described impact of change [EDF96], functionality distribution [RD97], and estimated development effort through use case points [HM95]. In additional to accommodating these measures, several other metrics are proposed to conduct the fit assessment.

Within the context of providing a fit assessment, transformations of various use case representations may be conducted as prescribed by the DEAT process model. The Chafi behavioral lifecycle object role stereotypes [Chaf96] are integrated into sequence diagrams. The state machine transformation technique from [ML97] and [Glin95] are also accommodated. Beringer's regular expressions are used for representing the use case dialogs [Beri97].

Mainstream object-oriented analysis and design techniques have generally proven inadequate to address all of the complexities of evolving domain specific architectures. Managing domain-specific architecture evolution is a complex and ill-defined problem. An early assessment of the stability and practical boundaries of the problem domain is needed in order to determine an appropriate level of analysis and to estimate the overall effort required for maintaining and evolving the architecture. Given a set of use case formalisms that represent an underlying business domain-specific architectural model, this thesis provides an approach to identify and utilize software engineering tools and techniques that address two primary challenges. The first challenge is to bridge the cognitive gap between management and software engineers. This is accomplished through the deployment evaluation sub-model of the DEAT process model in chapter seven. The second challenge is applying domain knowledge, design principles, standards, and programming skills to generate an application and to evolve the framework in such a way that desirable properties are maintained. These properties are identified in the domain engineering conceptual model from chapter four and the changes are enacted through the domain normalization operations that are prescribed as a result of the use case formalism and transformation sub-model of the DEAT process model.

As we add new applications through the domain architecture, we need to ensure that they are not destructive to each other. We also need to make certain that as we expand the domain model to accommodate new applications, that these future applications are not unnecessarily constrained or complex. As the next four chapters demonstrate, the DEAT process model addresses these concerns through the support of domain normalization operations, adaptive use cases, and parameterized business rule patterns.

Table 3.1. Use Case Formalism Focus (page 1 of 2)

	Static						Dynamic							Policy			Process			
	Data	Inheritance	Aggregation	Dependency	Refinement	Context	State	Events	Transaction	Actions	Contract/Pattern	Roles	Agents	Rules	Exceptions	Composition	Evolution	Parameterization	Users	Tools
Use Case Components [JGJ97]		✘	✘	✘	✘		✘	✘		✘	✘	✘				✘	✘	✘		
Use Cases Goals [Cock97]		✘	✘	✘	✘	✘						✘								
Adaptive Use Cases [Hurl97]	✘	✘	✘	✘	✘	✘	✘	✘	✘	✘			✘	✘	✘	✘	✘	✘	✘	
Enhanced Scenarios [Beri97]		✘	✘	✘	✘	✘	✘			✘				✘				✘		
Agent Based Use Cases [KMJ97]		✘	✘				✘		✘	✘	✘		✘							
Use Case Maps [Buhr97]		✘	✘						✘		✘							✘		
Change Cases [EDF96]		✘			✘					`							✘			
Task Scripts [HSFG97]		✘	✘				✘	✘				✘		✘	✘					
Use Case Classes [Jans95]					✘															
Use Case Formats [Harw97]						✘		✘			✘				✘					
Directed UC Graphs [KPW97]					✘					✘										
Scenario Trees [HSGK94]							✘	✘											✘	
Message Sequence Charts [AB95]		✘	✘		✘			✘								✘		✘		
State Machine Elicitation [ML97]							✘						✘							
Formalize UC/State Chart [Glin95]							✘	✘												
Synthesized UC Model [RKW95]	✘	✘	✘				✘			✘									✘	
Use Case Dialog Maps [Wieg97]							✘							✘	✘				✘	
Service Usage Models [KS94]		✘					✘	✘		✘								✘	✘	
Scenario Contexts [Zorm95]	✘					✘										✘	✘			✘
Scenario Strategies [CNM95]					✘						✘	✘	✘							
Model Large Bus. System [ABS95]				✘		✘									✘					
Episodes [RD97]				✘		✘		✘		✘								✘		
Use Case Reuse [Know95]					✘												✘			
Hypermedia/Visuals[ADP95]																	✘			✘
Hypertext Use Cases [Zeie95]																			✘	✘
Requirements Scripts [BB95]		✘	✘																	✘
Animating Use Cases[Thom95]																			✘	✘
Requirements Verification [HM95]																✘				
Formal UC Verification [Mend95]	✘									✘										
Component UC [DWMW96]				✘																

Table 3.1. (Continued – page 2 of 2)

	Static						Dynamic							Policy			Process			
	Data	Inheritance	Aggregation	Dependency	Refinement	Context	State	Events	Transaction	Actions	Contract/Pattern	Roles	Agents	Rules	Exceptions	Composition	Evolution	Parameterization	Users	Tools
Iterative UC Prototyping [Coll95]																✘			✘	
Action Workflow [MWFF92]								✘	✘	✘	✘				✘					✘
Workflow Mgmt System [Beed97]											✘	✘	✘							
Abstract Bus. Processes [GE97]											✘									
Organization Knowledge [FCS97]								✘				✘	✘	✘						✘
Key Event Dictionary [WF97]						✘	✘	✘												
Behavioral Lifecycle [Chaf96]								✘	✘		✘	✘								
Transaction Based Anal. [Raws95]									✘		✘									
Transaction/Event/Rule [Silv95]		✘	✘			✘	✘	✘	✘	✘				✘	✘					
Collective Behavior/Rules[KHT95]											✘			✘					✘	
ECA Business Rules [Herb95]						✘		✘		✘				✘						
GUIDE Business Rules [HH95]														✘						
Hierarchical Policy [Nass91]		✘										✘		✘						
ALBERT [DDB93]						✘					✘		✘							
Incremental Spec Evolution [JB93]		✘															✘			✘
Commitment-based SW[MTMS92]		✘									✘			✘		✘				✘
Baseline Reqts.Document [BuH89]																			✘	✘
OOD from Func. Specs [AP92]																				✘
Information Mapping [Horn89]	✘																		✘	✘
Problem Frames [Jack94]						✘														
Analysis Patterns [Fowl96]		✘	✘								✘	✘		✘						
Frame Technology [Bass96]														✘		✘		✘		
Collaboration Framework [DW97]					✘		✘	✘		✘	✘	✘						✘		
Collaboration Design [VN96]	✘	✘										✘								
Path Expression Groups [AC94]		✘	✘																	
Component Contracts [Holl92]													✘	✘						
Exception Handling [SSP95]															✘				✘	✘
Design Patterns [GHJV94]		✘	✘								✘	✘						✘		
Business Rule Elements [Ross97]														✘						

Table 3.2. Use Case Formalism Formats (page 1 of 2)

	Textual					Graphical					Dynamic			Other	
	Unstructured Text	Tabular	Structured Description	Formal Expressions	Script	Structure	State	Rules	Interaction	Implementation	Assembly	Visualization	Navigation	Role Playing	StoryBoard
Use Case Components [JGJ97]			✘	✘	✘	✘	✘		✘	✘					
Use Cases Goals [Cock97]			✘		✘	✘			✘						
Adaptive Use Cases [Hurl97]	✘	✘	✘	✘		✘	✘		✘	✘	✘	✘	✘	✘	
Enhanced Scenarios [Beri97]		✘	✘	✘		✘			✘					✘	
Agent Based Use Cases [KMJ97]			✘	✘	✘		✘		✘						
Use Case Maps [Buhr97]		✘				✘			✘		✘				
Change Cases [EDF96]				✘	✘				✘						
Task Scripts [HSFG97]			✘	✘	✘	✘	✘	✘	✘						
Use Case Classes [Jans95]						✘			✘						
Use Case Formats [Harw97]	✘	✘	✘		✘				✘						
Directed UC Graphs [KPW97]						✘			✘						
Scenario Trees [HSGK94]				✘			✘								
Message Sequence Charts [AB95]	✘					✘			✘						
State Machine Elicitation [ML97]					✘		✘							✘	
Formalize UC/State Chart [Glin95]				✘			✘								
Synthesized UC Model [RKW95]					✘	✘			✘						
Use Case Dialog Maps [Wieg97]			✘		✘										✘
Service Usage Models [KS94]	✘	✘					✘								
Scenario Contexts [Zorm95]	✘		✘		✘				✘						✘
Scenario Strategies [CNM95]									✘					✘	
Model Large Bus. System [ABS95]	✘	✘			✘				✘	✘					
Episodes [RD97]	✘					✘			✘	✘					
Use Case Reuse [Know95]										✘					
Hypermedia/Visuals[ADP95]					✘				✘			✘	✘		
Hypertext Use Cases [Zeie95]			✘										✘		
Requirements Scripts [BB95]					✘	✘			✘						
Animating Use Cases[Thom95]	✘				✘	✘			✘			✘			
Requirements Verification [HM95]	✘				✘							✘			
Formal UC Verification [Mend95]			✘		✘				✘						
Validate Component UC [DWMW96]					✘									✘	
Iterative UC Prototyping [Coll95]	✘				✘										

Table 3.2. (Continued – page 2 of 2)

	Textual					Graphical					Dynamic			Other	
	Unstructured Text	Tabular	Structured Description	Formal Expressions	Script	Structure	State	Rules	Interaction	Implementation	Assembly	Visualization	Navigation	Role Playing	StoryBoard
Action Workflow [MWFF92]					✘				✘						
Workflow Mgmt System [Beed97]						✘			✘						
Abstract Bus. Processes [GE97]					✘				✘						
Organization Knowledge [FCS97]	✘		✘		✘				✘		✘	✘			
Key Event Dictionary [WF97]		✘			✘		✘		✘						
Behavioral Lifecycle [Chaf96]						✘			✘						
Transaction Based Anal. [Raws95]			✘		✘	✘			✘						
Transaction/Event/Rule [Silv95]						✘	✘	✘	✘						
Collective Behavior/Rules[KHT95]			✘						✘						
ECA Business Rules [Herb95]						✘									
GUIDE Business Rules [HH95]			✘												
Hierarchical Policy [Nass91]	✘			✘											
ALBERT [DDB93]	✘		✘	✘	✘	✘			✘						
Incremental Spec Evolution [JB93]				✘	✘	✘	✘		✘		✘	✘	✘		
Commitment-based SW[MTMS92]				✘							✘				
Baseline Reqts. Document [BuH89]	✘										✘			✘	
OOD from Func. Specs [AP92]	✘			✘											
Information Mapping [Horn89]	✘	✘	✘		✘								✘		✘
Problem Frames [Jack94]			✘												
Analysis Patterns [Fowl96]						✘			✘						
Frame Technology [Bass96]				✘							✘				
Collaboration Framework [DW97]			✘	✘		✘	✘								
Collaboration Design [VN96]						✘			✘						
Path Expression Groups [AC94]				✘					✘						
Component Contracts [Holl92]									✘						
Exception Handling [SSP95]				✘	✘								✘		
Design Patterns [GHJV94]			✘			✘			✘						
Business Rule Elements [Ross97]			✘					✘							

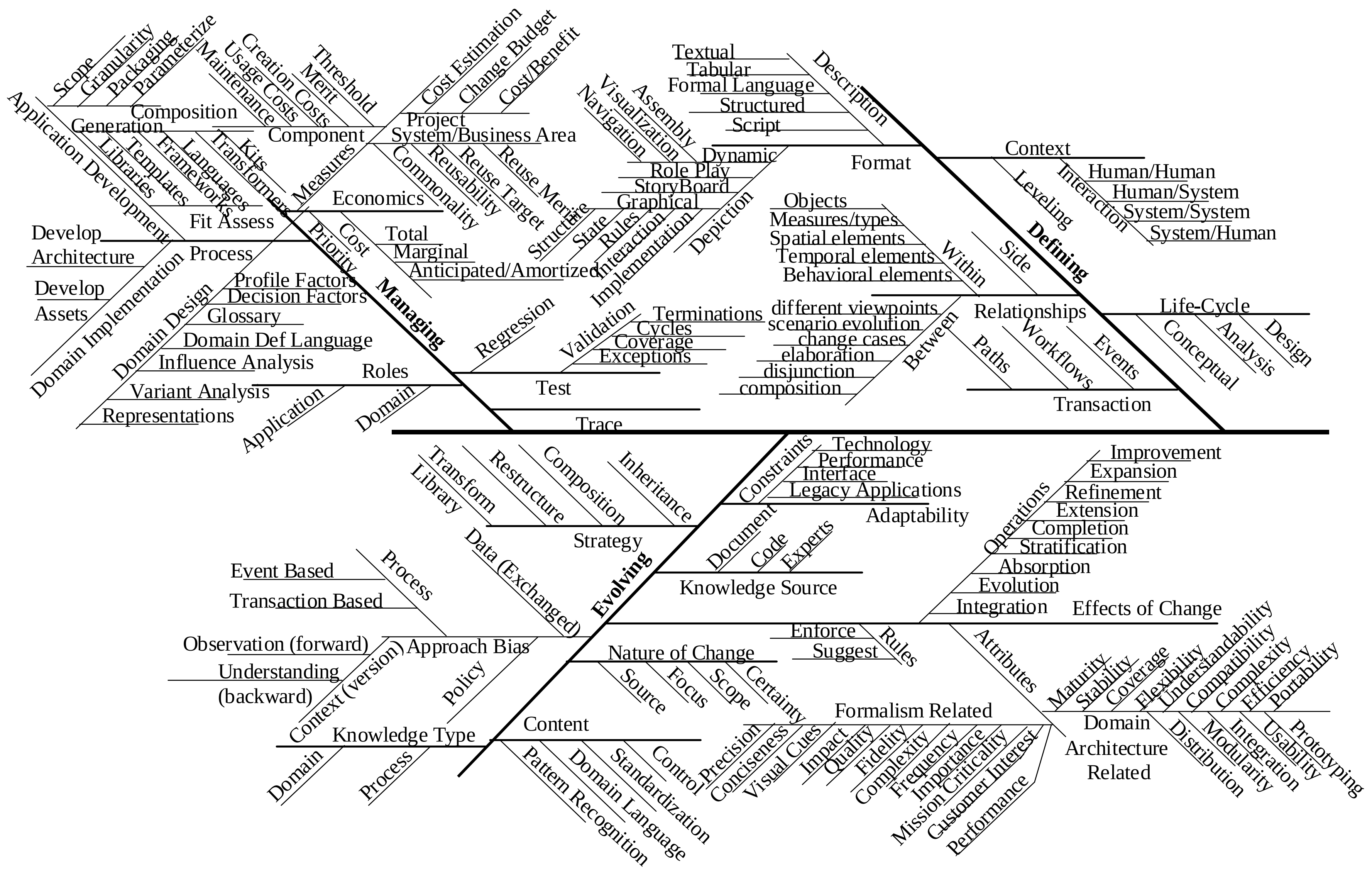


Figure 3.1. Use Case Formalisms Concept Diagram

CHAPTER IV

DOMAIN MODEL NORMALIZATION

Domain engineering is concerned with the development of an architecture for implementing a family of related applications. We need to make certain that as we evolve and expand the domain model to accommodate new applications, that these future applications are not unnecessarily constrained or complex. This chapter first presents a conceptual framework for managing the evolution of business domain models and their realization through reusable architectures. Each of the facets of the framework is discussed in the context of this research. *Development* related facets describe the type and structure of domain architectures. *Deployment* related facets elaborate application instantiation and evaluation issues. Operations that can be applied to a domain architecture are also introduced. Next, the concept of database normalization is extended to consider normalization of behavioral properties. Three aspects that are central to the notion of data normalization are examined to consider how they may be extrapolated to the larger context of domain model normalization: the separation of data elements into affinity groups, operational properties that eliminate data anomalies, and unbiased implementation. Having established a conceptual basis for domain normalization, this chapter proceeds to describe how operations defined in the conceptual framework are applied against the domain architecture to achieve normalization. The impact of structure, process, policy, and context normalization concepts is explored. Each operation's bias towards either order or chaos is examined in relation to its affect on components, use cases, parameters, and transformations. The chapter concludes with a presentation of first, second, and third domain normal form.

4.1. Domain Engineering Conceptual Framework

The domain engineering conceptual framework is intended to present a categorization of the most significant areas of concern regarding the field of domain engineering. Domain engineering, in this context, embraces all dimensions related to the development of domain models and the deployment of domain related applications. A four dimensional hyper-cube has been selected to graphically depict this conceptual framework. This representation has been chosen to stress the relationships among the various areas from which we can approach the field.

At the highest level, the domain model conceptual framework consists of six facets, or dimensions. They are captured in Figure 4.1 as the outside surfaces of a cube that has been arranged in a familiar flattened out pattern. This arrangement is intended to reinforce the notion that each of these dimensions is merely one perspective of viewing the practice of domain engineering. The topmost dimension box is labeled *Aspect*. It is considered the main entry point to classifying domain engineering. Immediately beneath it is the *Activities* dimension. This dimension is at the junction of all dimensions since it participates in both of the two primary axes: development and deployment. Immediately to its left is the *Type* dimension. It is placed in a left to right sequencing to demonstrate the necessity of selecting the focus of the domain engineering development process as a precursor to the life-cycle activities identified in its neighboring *Activities* dimension box. At the other end of this development is the *Style* dimension. This can only be determined after the first two activities have been tackled.

Along the deployment axis, the *Instantiation* dimension represents actual use of the domain model for creating applications. This is followed along the deployment axis by the *Evaluation* dimension. This final top-most dimension addresses compatibility issues with respect to domain model development and domain application deployment.

Continuing with the cube analogy depiction, another relationship can be established with respect to the complementary facets of the three-dimensional cube, i.e. those that are one opposites sides. *Type* and *Style* are paired in a specification-realization dichotomy along the domain model development axis. Likewise, *Aspect* and *Instantiation* are paired in a specification-realization dichotomy along the deployment axis. *Activities* and *Evaluation* are paired in a control-feedback relationship where the evaluation dimension represents a meta-level view of the overall activities of domain engineering.

Orthogonal to these facets is the normalization aspect of domain engineering presented in this chapter. Normalization is graphically depicted as the textured inner cube underlying the six top-most facets. Normalization includes the operations that are used to manipulate the domain architecture, the attributes that reflect its state, and the compositional rules that govern the normalization process. The normalization process affects the overall domain model through four primary views and their related constructs. The structural view is expressed in terms of components. The process view is expressed through use case formalisms. The policy view is manifested through parameters that define requirements. The context view captures the transformations that can be applied to the domain architecture. These four sets of views and constructs form the bridge that connects domain normalization to the development and deployment axes.

Domain engineering operations normalize the structure and behavior of the domain architecture. They represent the glue that holds this conceptual framework together to form a multi-dimensional representation of the field. Without these operations, the domain model cannot evolve and new applications cannot be implemented. These operations represent the major conceptual underpinning of this conceptual framework. They have profound impact on both development and deployment.

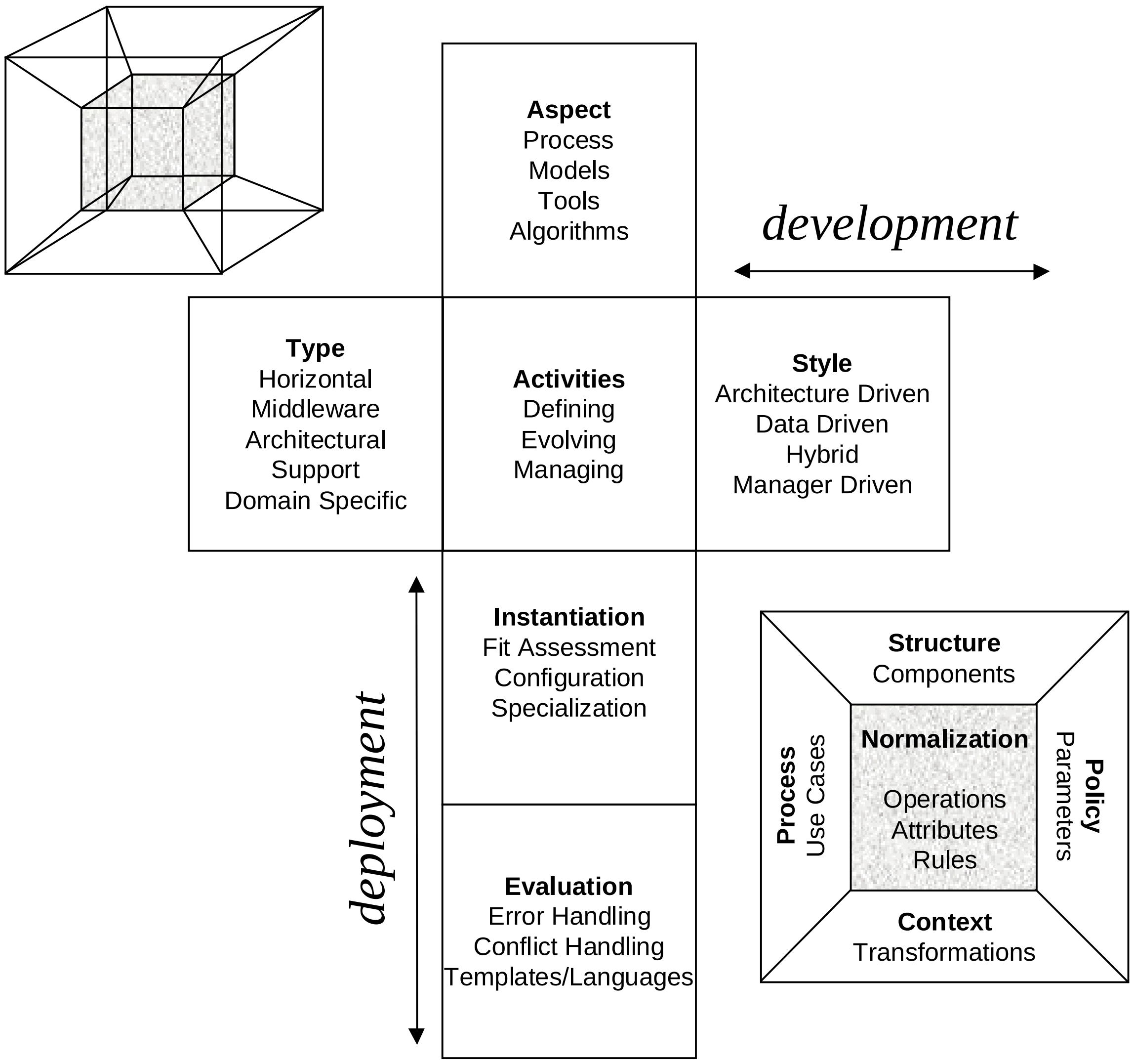


Figure 4.1. Domain Engineering Conceptual Framework

4.1.1. Domain Engineering Aspects

Domain engineering aspects are classified into four categories: models, tools, algorithms, and processes. The primary focus of this research relates to models. Models provide the means to reason about a business domain. Specification and analysis models allow us to capture the problem space. Others, such as design, implementation, and testing models, represent the solution space.

The transformation of models for one perspective to another typically requires a formal algorithmic process. Tools may be constructed to automate the transformation process by applying algorithms that transform models from one type to another, including transformations to executable code.

Organizational processes establish the context for dealing with a business domain. They represent the single most important aspect in determining the success of the domain architecture. Unless a well-defined process is implemented and followed, the domain architecture will suffer. Domain architectures are intended for a high degree of reuse. This reuse will only be realized by adherence to disciplined process management.

4.1.2. Domain Engineering Activities

Domain engineering activities refer to the definition, evolution, and management of a domain architecture. The definition of a business domain architecture may result from a new development effort. However, many organizations are in the situation of having already built several applications addressing the target business domain. The ability to take a project that exists only in code and reverse engineer the first cut at a model from the information extracted from that code base can be critically important to such organizations. A focus on conceptual analysis may yield corporate views and classes that can be reused throughout the organization. A focus on design may yield more flexibility in the reverse engineering process. The robustness of transformation algorithms and automated tool support will ultimately influence the value of the reverse engineering effort.

Even with the best tool support, a critical aspect of the reverse engineering effort is the need to create use cases and sequence diagrams at the proper level of conceptualization. Pattern mining, which is the recognition of patterns in existing code, could also be used to raise the abstraction level. This can only be accomplished if at least some domain engineering has been accomplished to establish the fundamental concepts.

The evolution of a domain architecture is accomplished through several operations that can be applied to an architecture. These operations allow the architecture to grow and mature in an orderly manner. A more detailed discussion of the nine architecture operations will be covered later in this chapter.

Managing the evolution of the business domain architecture focuses on change impact assessment. This is critical because change propagation, left unmanaged, can result in multiple versions of the same domain architecture artifact. In order to reuse an artifact, some form of certification is needed, such as a reuse contract between managers and users of the reusable asset [SMLD96]. Structured documentation is needed in the form of rules that indicate where and how to test and adjust applications to facilitate propagation of changes. This activity is covered in greater depth in chapter seven.

4.1.3. Domain Architectural Types

Frameworks are often used to implement the functionality for a particular domain, but this is not necessarily the case. A framework represents one possible architecture for a domain, but other types of architectures are possible. However, an underlying assumption of this research is that a business domain architecture will be modeled as a framework of reusable, collaborating components.

Five classifications of domain types are defined. Three of these cut across many business domains, while addressing the same problem in each. Interface domains are user-visible frameworks such as a graphical user interface or voice response system. A middleware domain provides inter-process communication, such as database connection software. Support domain deal with the lower level aspects of systems, such as file access and device drivers [Adai95]. These are not of concern in this thesis and will not be discussed further.

An architectural domain is a higher level framework that subsumes all of the previously described frameworks. In addition, it incorporates operation, administration, and maintenance components [BDRV91]. Workflow management engines can be considered an architectural framework.

The last domain type is the business-specific domain. This type is the primary focus of this research. Domain specific frameworks are generally built on top of an infrastructure framework. They contain object libraries and black-box components that model the major entities, processes, and interfaces in a domain. They are generally business, or server, objects as opposed to interface, middleware, or support objects. Unlike configurable applications, domain specific frameworks are targeted at the software developer and not the end user [SS97].

Domain specific frameworks follow the same principle as domain independent frameworks, except that the business objects have more knowledge about the particular application domain than generic objects. This facilitates development of a specific application because configuring a business framework is easier than configuring a generic framework to build the same application. With more powerful configuration tools, the domain specific framework approaches the ease of a configurable application without sacrificing the flexibility.

4.1.4. Domain Architecture Styles

Several architectural classifications have been proposed. Examples of architectural frameworks include layered, pipes and filters, presentation-abstraction-controller, reflective, microkernal, repository, broker, abstract machine, and client-server [BM95]. Rather than enumerate each possible type, a simpler classification scheme is presented. These structures may be classified as either architecture-driven, data-driven, hybrid, or manager-driven. Architecture-driven structures are primarily inheritance based. Data-driven structures are composition based. Hybrid structures exhibit an even bias towards both inheritance and composition. One final structure classification is provided. However, it is orthogonal to the others. The manager-driven architecture deals with a controller, which is a common style for business domains that rely on workflow oriented processes.

When we speak of software architectures, the classifications generally are made at the level of granularity that describe the general function of the system. What is needed is a sense of architectural style at each of the levels of the system. For example, if we want to build a house, we can choose among a variety of styles, such as a ranch, Victorian, or Georgian. The function of the house hasn’t changed, just the way that the function has been implemented. When a Georgian home is built, certain attributes of the home are expected to be there: a certain appearance, traffic pattern or floor layout, and appropriate landscaping. A generic architecture specifies that we will build certain features such as walls, rooms, baths, porches, and windows. But it is the framework, or interactions between each of these entities that define the style. These recurring interactions are the underlying theme behind the pattern movement.

A pattern defines a micro-architectural style. Consequently, a pattern language defines an overall architectural style. However, just as certain building styles shouldn't be mangled, neither should certain patterns. Otherwise, the contrasting architectural styles will affect the level of interoperability that can be achieved between components. When stylistic differences exist between two components, they run a risk of complications when attempting to bridge their respective protocols [PA95]. For example, a component that relies on being notified of events will have difficulty with a component that assumes a polling style of event notification.

4.1.5. Domain Application Instantiation

The instantiation of an application from the domain architecture is primarily accomplished through configuration and specialization. Configuration of an application is facilitated through the use of parameters. When existing components cannot be successfully configured to meet the requirement specifications of the new application, the new features are typically integrated through specialization of existing components. In order to make such a determination, a fit assessment should be performed that evaluates the level of changes and estimated effort that will be involved in delivering the new features.

The basis for the fit assessment is directly tied to the adaptable and adaptive nature of the domain architecture. *Adaptable* software is defined as software that is easily changed, whereas *adaptive* software is software that is automatically changed according to context [Lieb95]. It is difficult to measure changeability. Moreover, performance constraints or requirements of backward compatibility further complicate measurements.

4.1.6. Domain Model Evaluation

Evaluation of a deployed application from the domain architecture completes the primary facets of the conceptual framework model. The primary focus of the evaluation facet within this framework is error and conflict handling. They provide a valuable measure for assessing adequate coverage of features. In addition, a retrospect of the fit assessment should reveal opportunities to evolve the architecture. Initiatives to develop new templates and extend domain specific languages may emerge as a result of the evaluation.

The complexity of an adaptable domain model is an impediment to its maintenance and evolution. Moreover, heavily parameterized code increases code complexity that ultimately leads to diminishing returns. It may be easy to adapt artifacts due to automation, but it is difficult to create them in an adaptable form [Davi92]. In order to promote adaptability and overcome these difficulties, a domain specific language must evolve along with the domain architecture. This language must be able to express functionality independent of internal structure. Domain specific languages permit machine-processable representations of process and policy views of a common underlying domain model. Once captured in this form, they can be used to generate and evolve structures through mechanisms that express customizations of the domain architecture's generalized facilities.

4.2. Extending the Concept of Data Normalization to Domain Models

Normalization is the process of converting an arbitrary relational database design into one that will have "right" relationships between them. This is accomplished by separating data elements into affinity groups. The term normalization is derived from the Latin word *norma,* which was a carpenter's square. When two lines meet at a right angle, they are said to be normal to each other [KL95].

Through the process of normalization, a database design is produced that minimizes data anomalies and inconsistencies. The resulting structures can then be manipulated in a powerful way with a simple collection of operations. Usage is not a consideration in the normalization process. In other words, the size of database tables or the types or frequency of queries against those tables are not a factor [Gold85]. However, both normalization and usage patterns are taken into consideration when making explicit tradeoffs between efficiency and integrity.

In order to extrapolate these normalization concepts to a domain model, a few of the key aspects just described are analyzed in detail. These are the separation of data elements into affinity groups, operational properties that eliminate data anomalies, and unbiased implementation.

4.2.1. Affinity Groups

If data normalization separates elements, such as names, addresses, or skills, into affinity groups, then behavior normalization should separate functions into similar types of affinity groups. The types of affinity groups that may be appropriate for functions are ones such as derivations, setting an attribute value, and propagating a message. These functional affinity groups are orthogonal to the data elements. If we create a matrix that places data affinity groups along one axis and the functional affinity groups along the other axis, it becomes possible to identify usage pattern between these two affinity groups. For example, all data elements would require some function to set its value. Furthermore, the particular method or style of setting the data element values would typically be consistent throughout the application, and possibly for the complete family of applications. As such, we can consider each application instance an affinity group along a third axis. Figure 4.2 illustrates the repetition of usage of the data attribute with the 'set' function affinity group across applications created from a shared domain architecture. Guidance as to how these intersecting affinity groups can be combined to create an application relies on composition rules. These composition rules are contrasted with business rules, which apply to the subject domain being modeled.

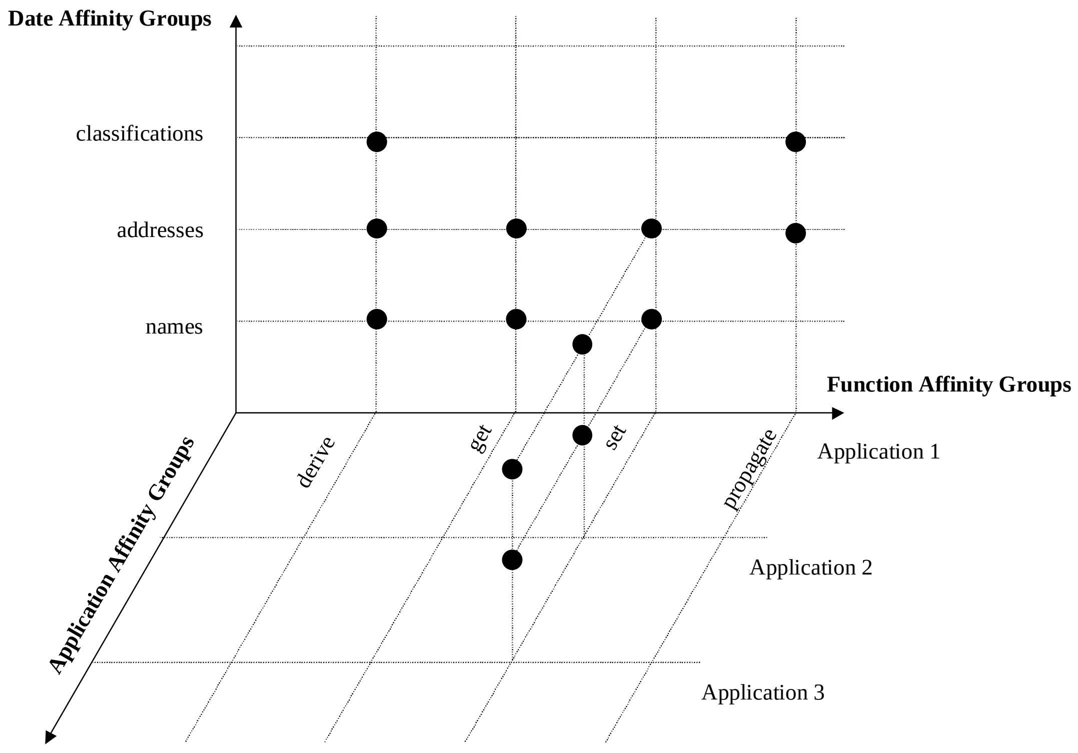


Figure 4.2. Affinity Groups for Domain Normalization

4.2.2. Operation Properties that Avoid Anomalies

Domain normalization should be able to convert an arbitrary domain model into one that will have good operational properties. The relational database model stands in contrast to the hierarchical and network database models by its nature of being solidly grounded in mathematical theory. Because of the ability to couple procedures with data, the same capabilities that are provided in the relational model are available within a domain model. Furthermore, addition semantic information is available that will allow relaxation of normalized data structures through the use of attached behavioral properties. Thus, database models that would have otherwise been considered denormalized, can still be represented as a normalized domain model through behavioral constraints such as triggers and stored procedures. Relaxing certain data structures realizes a balance between complete data model normalization at one extreme and complete data encapsulation through stored procedures at the other extreme.

One objective of domain normalization is to produce a domain model that can be manipulated in powerful ways with a simple collection of operations while minimizing the data and behavioral anomalies and inconsistencies. Whereas the data model operations are intended to yield data from a database, the domain model operations are intended to yield executables from a domain architecture or framework. When considering operations on a domain architecture, the entire collection of objects is treated as data. This represents an intensional view of the domain model, since at this point the objects are treated as data only, not as elements capable of manipulating other elements, such as data.

A set of database tables, independent of any domain knowledge, is insufficient to complete normalization beyond second normal form. To proceed to third normal form, we need to consider functional dependencies. Knowing that there are no transitive dependencies carries a measure of semantic information. But another concern is whether of not there is sufficient semantic information to provide appropriate relationships among classes. Many times the decision has seemed arbitrary with respect to the creation of class hierarchies, composites, or some weaker relationship among object types. Domain normalization needs to be defined in such a way that certain semantic properties will be preserved, certain anomalies can be avoided, and certain class operations that are independent of any specific application instance of the domain model can be performed. In order to adhere to these constraints, three categories of operations are defined.

1. *Intensional operations.* Operations that are performed on portions of the domain architecture.
2. *Extensional meta-operations.* Operations that are performed by the entities within an application instance that either inquire about or manipulate the structures and protocols of collaborating entities.
3. *Extensional operations.* Operations that are performed by the entities within an application instance that fulfill the expected functionality of the application.

If, as in data normalization, one of the primary objectives of domain normalization is to avoid certain anomalies, then we need to precisely define what anomalies can be present in the domain model. By first considering data modeling, the *update*, *delete*, and *insert* anomalies each refers to repeating data that is in either first normal form or second normal form. When certain fields are repeated, updating one row puts it out of synch with the others. Similarly, the *delete* anomaly ignores the separation of the concept behind the repeated field from the non-repeating attributes. For example, a library book (with attributes of title, author, and publisher) is separate from the collection of instances of that book (with attributes of catalog-number, copy-number, borrower-id, and due-date). The *delete* anomaly arises when deleting all instances would also destroy the book information. The *insertion* anomaly addresses the inverse of this situation, i.e. when information cannot be inserted, such as the book information, because no copies exist.

When relating these anomalies to domain models, updating an attribute or method typically relies of factoring of functionality to the highest possible level in a class inheritance hierarchy or delegated to the same class as a member of an aggregation hierarchy. If an attribute is propagated through a complex class graph, then many classes may be affected by the deletion or addition of an attribute. Adaptive software deals with this problem through their propagation patterns [Lieb96]. But propagation patterns don't address derivation functions that might not be needed. Subject oriented programming comes closer to pruning functionality, but assumes a more rigid data structure [OH95].

Delete and *insert* anomalies at the domain normalization level refer to the creation or removal of types or classes. If we remove all collaborations that refer to a role/class then the class disappears. Similarly, including a role/class in the domain model that does not yet have any collaborations essentially renders it meaningless. In order to avoid this, we must be able to maintain a class that captures information regarding what roles it can play. Just as we were able to preserve the book information by creating a new entity, we must preserve the entity information by creating a new meta-class to hold the class until needed. The join property is based on the collection of roles that it can play, and just like in relational algebra, the selection can be any one from the candidate set. Thus, one of the fundamental principles of domain normalization is that all classes/types must belong to a meta-class in order to be in normal form.

4.2.3. Unbiased Implementation

One outcome of domain model normalization is to provide constructs that are not biased towards any specific application instance. Unforeseen queries in a normalized data model, such as the ad hoc nature of decision support systems, can be handled in an unbiased way. The data normalization process does not consider the type or frequency of these queries. It also does not take into consideration the numbers, either absolute or relative, of particular entities or relationships. Consequently, domain normalization should remain unbiased towards particular types of applications. In other words, the domain normalization process should not consider the numbers of classes or collaboration patterns, or what types of applications are generated. It is not tuned for a particular business domain framework.

This does not mean that performance strategies can be ignored. Consider relational data model designs that include disk strategies or needed indexes. These are not part of the logical design, but perform important needs with respect to deployment. In a data model, performance considerations may force changes to the logical design, such as restructuring of the data tables for warehousing or replication strategies. As previously introduced when discussing operational properties, domain modeling can permit a data structure to become denormalized, while still preserving the desirable properties of normalization. By attaching behavior to data structures, domain model normalization allows alternative implementations that remain unbiased to a particular implementation by maintaining semantically equivalent structures of data and behavior as alternate representations. Since most database vendors provide triggers and stored procedures, domain normalization can be preserved by coupling certain triggers with denormalized tables to maintain semantic equivalency. Composition rules, as a form of meta-rule, can assist in determining when triggers coupled with denormalized tables are appropriate over normalized data structure.

As we can derive from the preceding illustration, normalization allows us to evaluate explicit tradeoffs between efficiency and integrity. It should not be followed blindly. We will be exploring the implications and properties required for implementing such rules in next section.

4.3. Domain Model Normalization

Just as in data model normalization, there is a clear distinction between the steps needed to get the domain model into normalized form and those operations that can be performed on a normalized domain model. In deriving an essential set of operations for domain model normalization, four distinguishing aspects of a domain model are established. These aspects are parameters, components, transformations, and use cases. These four aspects were selected because of how they interplay with the fundamental principles of domain architectures that form the basis domain normalization, as presented below:

1. Domain growth is chaotic.
2. Domain maturity is orderly.
3. Domain development, in absence of application development, tends to foster maturity, and thus tends to be orderly.
4. Application development, in absence of domain development, tends to foster growth, and thus tends to be chaotic.
5. Domain development, in conjunction with application development exhibits opposing tendencies towards chaos and order.
6. After the initial application is developed, domain development is primarily concerned with maintaining order.
7. Requirements for new application development are the driving force behind domain growth.
8. Domain growth is a direct cost to application development.
9. Domain maturity reduces the marginal cost of application development.
10. Absorbing new application development into the domain architecture is a reactive approach to domain growth.

The significance of these principles leads to the following observations. First, a domain architecture that is used for application development will tend to grow and mature. Use cases will expand both in number and in elaboration of alternate path scenarios. Generation of applications from the domain architecture will increasingly rely on parameterization and domain specific languages to manage the increasing complexity of more components. Behavior preserving transformations will be relied upon to provide the flexibility necessary to respond to application development demands and domain architecture restructuring. Furthermore, domain development will be primarily driven by marketing and economic considerations regarding future application development.

The level of normalization achieved by a domain model is derived from a collection of interrelated subordinate normalization views, each on which corresponds to one of the four distinguishing aspects of a domain model (parameters, components, transformations, and use cases). These views are data normalization, process normalization, policy normalization, and context normalization. Overall domain normalization is determined through normalization levels or each its component levels. For example, a domain model can be in data third normal form, yet not be normalized. In order to determine the level of normalization for each constituent level, the impact of domain normalization operations on each aspect must be explored.

4.3.1. Use Case Formalisms

Use case formalisms are the most versatile and essential aspect of domain normalization. The functional characteristics of use cases provide the closest analogy to data normalization and its rigorous relational algebra base. A use case represents a view, or projection of functionality of the domain architecture. A scenario represents an actual instance of that view. Thus, similar to the relational tuple, a scenario captures the persistent state of that functionality. However, it has the capability of extending far beyond capturing data structure by maintaining semantic information with respect to when, how, and why the scenario was invoked. It could be argued by data-centric purists that this information is merely additional tuples that are captured in a relational database. However, it is the semantic coupling of process, policy, and data within the domain model that makes this more than the sum of its persistent parts. The ability to reason about this data through coupled behavior is the critical distinguishing feature that separates domain modeling from data modeling within the context of model normalization.

Just as a database matures to contain more data, a domain architecture matures to contain more scenarios. Scenario growth constitutes a vertical expansion of the domain. Additional scenarios fill out a more complete set of alternative paths that may be taken. Similarly, just as additional tables or columns may be added to the database to include more information, additional use cases may be added that provide more features, constituting horizontal growth of the domain model. For purposes of domain model operations the vertical growth is referred to as an *expansion* operation, while the horizontal growth is referred to as an *integration* operation.

Use case formalisms are primarily concerned with process normalization. At some point in the normalization process, all scenarios map to a set of components. This mapping is exemplified by weak coupling between components in the form of association and collaboration object-oriented constructs. Such constructs include delegation, instantiation, navigation, accessors, notification, externalization, and negotiation. Although each use case step can be parameterized, it would be more practical to identify use case dialog splice points where several use case alternate paths originate or converge back into the main path. Normalization typically requires a use case to begin and end with the same step/state and each step must be an atomic indivisible function. Exception handling, by definition, is not required to conform to these influences. Algorithms for minimization of event-states and simplifying use case dialog regular expressions provide the mechanism for identifying splice points.

The decomposition of processes into normal form deals with several interrelated properties. Each use case step represents an atomic action. This action can either be completely independent or it can be dependent on another action. If it is dependent upon another action, then the dependency can be either in the form of a *controlling* action or in the form of an *influencing* action. Likewise the action itself can exert a controlling or influencing effect on yet another, different action. Such influence is referred to as an *interfering* action. Such interference is typically expressed through temporal logic. For example, a blocking action would stipulate that if one action is currently in progress, then another action couldn't occur. This is referred to as a *prevention* action [DDB93]. Another interfering action is an *obligation*, which stipulates that an action has to occur if the conditions are true. An obligation becomes an *exclusive obligation* if the action can occur if and only if the conditions are true. The full range of interfering actions provide for constraints with respect to imposed serialization, parallel execution, and decision preservation. Decision preservation with respect to use cases involves scenario determination. A use case process must be known a priori, however the specific scenario typically is not known. It can only be probabilistically determined. Decision preservation unambiguously captures the identity of the actual scenario instance invoked as well as providing an explanation facility for the scenario selection.

4.3.2. Parameterization

As functionality increases in terms of the number of use cases, the number of parameters needed to configure a system will also tend to increase. In order to balance this trend, parameters must be generated that capture more information from choices. Focus must shift from similarity to differences. This is accomplished though the concept of refinement. As a domain architecture is utilized to implement many applications, patterns emerge regarding similar configuration choices that have been made. These collections of choices can be aggregated into refined architectures in order to simplify the choices available. Thus, by automating the process of managing similarity, the user can focus only on differences in configuring a system from the domain architecture. High level choices must be capable of rapidly configuring large numbers of components without the burden of having to deal with minor variation.

New functionality increases parameterization. Conversely, new applications combined with domain architecture operations will reduce parameterization. The former is referred to as an *extension* operation, the latter as a *refinement* operation. This is a fundamental definition of normalization – i.e. the reduction of complexity and inconsistency.

Parameterization is primarily concerned with policy normalization. Policies are represented formally through event-condition-action tuples that express structural and behavioral assertions of the domain model. The normalization process identifies and isolates them in such a manner that parameters can be attached to each rule at the component level and therefore influence behavior within a component. Similar to use cases, rules are structured in template form. Furthermore, a collection of rules can be organized into a decision tree format. Parameterization of rules does not preclude implementing policy as functional logic within the domain architecture. Custom interpreted code that can parse and execute rules dynamically is still possible. However, this strategy represents a denormalized implementation of rules that is appropriate in limited contexts such as prototyping. Externalization and meta-object queries can also be used to dynamically attach parameters to components in a manner that provides a greater degree of normalization.

4.3.3. Transformations

Two basic types of transformations exist – those that increase the domain of possible representations and those that reduce the number or representations. The former is referred to as *restructuring* transformations. The latter is referred to as *evolving* transformations. Evolutionary transformations are one measure of a domain model’s maturity. Typically, this means progression from white-box, inheritance, and generative based approach to a black-box, aggregation, and compositional based approach.

Transformations are primarily concerned with actual deployment of applications from the domain architecture. More precisely, transformations can change deployment to different platforms. Although restructuring and evolutionary transformations deal with design issues, such issues are only considered within the context of creating an executable. When methods and data are factored into parent classes through transformations, we focus on object-oriented concepts that exhibit strong coupling among classes usually found in inheritance and aggregation hierarchies. In particular, the use of generalization, specialization, overriding, extension, overloading, and restricting tends to create deployable modules that vary along a continuum of mutability that can range from very stable to highly volatile. As evolutionary transformations increasingly utilize delegation in lieu of inheritance, adaptable and adaptive software techniques allow closing off the component which can significantly alter the contents of deployable modules. Many of these changes are in the realm of domain architecture meta-operations such as abstracting, migrating, refactoring, and encapsulating. Hence, transformations foster context normalization by emphasizing maturity and stability in the partitioning of deployable modules through discrimination of the stable and variant parts of the domain architecture. Context normalization ultimately represents normalization of classes, processes, and policies across application instances. It touches upon the areas of version control, configuration management, and environment space [BK86], [KAC86]. Through context normalization, we can create any version of an application.

4.3.4. Componentization

Components play a crucial role in domain architecture since they form the encapsulated modules of reusability. Again there are two forces at work when dealing with components. The first is the trend towards framework *completion.* This means the filling in of variable behavior with pluggable modules. The second opposing trend, referred to as *stratification*, segregates existing functionality into smaller modules of reusability to address finer grained specifications. This opposing trend recognizes the need to provide additional variable, or hot spots, where behavior was deemed to have been invariant or well defined. This type of domain architecture operation represents the “semantic earthquake” that shakes up an existing perspective or visions of the domain that no longer adequately provides a rich enough description of the real world that it is modeling.

Componentization is primarily concerned with structural normalization. The simplest component consists of a single object. Taken to the extreme of decomposition, all attributes, functions, and rules can be defined as their own objects. However, for practical purposes, we consider such decomposition to constitute "sub atomic particles" that should not exist in isolation within the domain architecture. We draw an analogy to protons, electrons, and neutrons of basic attribute objects. As part of our definition of basic component building blocks, we apply the Law of Demeter [LHR88]. This principal requires that an object (which for our purposes extends to its class and/or type) may not have knowledge of any other objects that are not explicitly defined as member attributes or as arguments to its methods. Although a distinction among the concepts of composition, aggregation, and association is not required, the general rule is that member attributes should be reserved for aggregation and composition, while method arguments should be reserved for collaborative associations.

Although it is conceivable to create basic objects that do not contain any data attributes, the most dominant and useful primitive atomic object is the data attribute. It encapsulates the storage of a value that is made available through accessor functions. In addition, rules may stipulate valid values, presentation, and authorization for the data attribute. Thus, as elementary as this type of object appears to be, it still represents both of the basic structural properties: encapsulation and separation of three basic responsibilities [Hend97]. Encapsulation requires the elimination of all global variables. Responsibility for *knowing* also requires that all data be encapsulated in objects that conform to a class/type definition. Responsibility for *doing* requires that accessor methods be utilized to set and retrieve values. Responsibility for *enforcing* requires that validation be performed by the attribute object.

4.3.5. Normalization Operations

We have shown in this section that domain normalization operations transform a domain model into normalized form through interaction with four basic domain modeling aspects. Each of these aspects exhibits a tendency either towards chaos or towards order. Whenever the domain model grows, the tendency is towards chaos. The danger lies in too much growth too quickly. In such an environment, chaos can overwhelm to domain model to the point that it becomes unusable. At the other end of the spectrum, a maturing domain model tends towards order. Similar dangers also lie in this extreme. Too much order is symptomatic of a problem domain that is either too small or too unappealing to warrant attention. The driving force here is to consider the fractal nature of domain modeling. Smaller domains should become integrated into larger domain through integration operations. The problem space of a mature domain should be subsumed by the problem space of progressively larger domains until the concerns of the entire enterprise are addressed. At this level, the domain model and the enterprise model are merely different view on the same problem space. Table 4.1 summarizes each of the domain normalization operations as they relate to the four domain model aspects. This is followed by a more thorough description of each normalization operation.

Table 4.1. Domain Normalization Operations

		Operation Bias	
Mechanism	Aspect	Order	Chaos
Use Case Formalisms	Process	Expansion	Integration
Componentization	Structure	Completion	Stratification
Parameterization	Rules	Refinement	Extension
Transformation	Context	Evolution	Reconstruction

- <u>Expansion – Increase number of scenarios – (use case depth)</u>. The *expansion* operation adds new features to a domain model by providing distinctive new scenarios to existing use cases.
- <u>Integration – Increases number of use cases – (use case breadth)</u>. The *integration* operation adds new features to a domain model by increasing the number of use cases that directly interact with actors at the boundary of the domain being modeled. These additional use cases are the result of either merging previously separate domains together or by adding new use cases to the problem space.
- <u>Refinement – Increase descriptive power or parameters</u>. The *refinement* operation creates a more specific framework by clustering sets of related components together that can be recalled by a single parameter or domain specific language construct. The intent is to reduce the quantity of decisions required to implement an application from the domain architecture.

- Extension – Increases number of parameters. The *extension* operation creates a more generalized framework by introducing parameters or variability through domain specific language constructs. The intent is to increase the scope of the problem domain by leveraging existing components to perform more generic functionality at the expense of more effort to configure the components.
- Evolution – Increases stability of components. The *evolution* operation creates a more mature domain architecture by sealing off variability normally provided through white-box, inheritance based framework mechanisms. This is done through transformations that replace these mechanisms with black-box, aggregation based framework mechanisms.
- Reconstruction – Increases volatility of components. The *reconstruction* operation creates a more volatile domain architecture by increasing the number of alternative representations or implementations that can be selected to implement a given application as the result of behavior preserving transformations.
- Completion – Increases number of components (breadth). The *completion* operation adds concrete components or modules that provide the full range of functionality needed for a predefined hot spot in the domain model, thus potentially reducing the amount of new code that would have to be generated for any new application.
- Stratification – Increases number of components (depth). The *stratification* operation partitions existing concrete components or modules by defining new hot spots where none previously existed, in effect adding additional layering of domain subsets that can be selected or customized to implement an application.
- Absorption. The *absorption* operation is a composite of the other operations. It reintegrates instantiated frameworks or applications with the domain model. This is typically the result of independent project efforts to deliver an application that are later identified as candidates for reuse by the domain model and architecture.

4.3.6. Normalization Attributes

The performance of any domain model normalization operation will be reflected in the model's state, which is captured through a collection of nine attributes. Each attribute defines an enumerated domain that provides an indication of its relative measure with an arbitrary scale. The domain for each attribute may be quantified on a scale from zero to five in a manner similar to the Capability Maturity Model [PCCW93]. However, further elaboration of this measurement is beyond the scope of this thesis.

- <u>Maturity.</u> The *maturity* of the domain architecture is directly influenced by the *evolution* and *completion* operations. It measures the progression of the domain from an inflexible one-of-a-kind implementation to an extensible framework and ultimately towards a fully configurable application generator.
- <u>Stability.</u> The *stability* of the domain architecture is similar to its *maturity* and thus is also directly influenced by the *evolution* and *completion* operations. However, it differs in the sense that *integration* operation may continue to be performed that have no significant impact of the characterization of the domain model by the *maturity* attribute, but indicates that the domain model is still undergoing significant change.
- <u>Coverage-scope.</u> Three operations affect the *coverage-scope* of the domain model. These are the e*xpansion*, *integration*, and *absorption* operations. As more features are added by these three operations, the coverage and scope of the domain model tends to increase. However, the integration of an immature and relatively incomplete domain model can reduce the overall coverage even though the scope has increased.
- <u>Flexibility-resiliency.</u> The *reconstruction* operation directly increases the *flexibility-resiliency* measurement of the domain architecture since the number of alternative representations or implementations that can be selected increases.
- <u>Understandability.</u> The *understandability* of the domain architecture is an indirect measurement that is a function of the domain architecture's maturity, scope, complexity, and adherence to standards.
- <u>Compatibility-standardization.</u> The *compatibility-standardization* attribute has no direct connection to normalization operations. Rather, it measures the potential for growth through acquisition of third party and off-the-shelf domain model extensions. As a result, it may also influence the understandability of the domain model.
- <u>Complexity.</u> Four operations directly influence the *complexity* measurement of the domain architecture. The *refinement* and *completion* operation will tend to reduce complexity, while the *stratification* and *extension* will tend to increase complexity.

- Efficiency. The *efficiency* of the domain architecture is an indirect measurement that may be influenced by any or all of the other attributes. It serves as a measure of the effort required to implement an application from the domain architecture.
- Portability. The *portability* of the domain architecture is an indirect measurement that is a function of domain model support tools to take machine-processable representations and implement them in different deployment environments.

4.4. Domain Model Normal Forms

The presentation of domain model normalization begins by examining what normal form means to a relational database and then extending this notion to an object-oriented model. From there, an extrapolation is made as to what constitutes domain normal form. Normalization for object-oriented class structures differs from relational database in several ways. However, corresponding concepts can be found that directly relate one to the other. Relational databases are concerned with relations, tables, tuples, primary keys, and foreign keys. Object-oriented domain models are correspondingly concerned with types, classes, objects, object identity, and associations between classes. Relational databases have several build-in facilities that provide default implementations that must be explicitly declared in a class model. Default sorting order rules for a table column are declared at the server level. Classes can duplicate this functionality by explicitly defining equality and less-than functions. Table 4.2 details the steps that would translate an object model into third normal form.

Table 4.2. Applying Normal Form to an Object-oriented Model

Form	Data	Object
1NF	Move data into separate tables. Define a primary key.	Move objects into classes. Creating object instances that each have a unique object Id. Define equality and less-than test methods that conform to a candidate key.
2NF	Separate data that's only dependent on a part of the key.	Factor into different class attributes that are only dependent on part of the candidate key; create an association.
3NF	Separate date that doesn't depend solely on the primary key.	Factor into a different class attributes that exhibit transitive dependency; create an association.

Our definition of domain normalization consists of four aspects – structure, process, policy, and context. In one respect, domain normalization measures the maturity of a business domain-specific architecture. In contrast to maturity models, such as the Capability Maturity Model [PCCW93], domain normalization is only concerned with properties of the domain model itself, not with the process of using it. The significance of understanding the normalization level of a domain architecture comes into play when performing fit assessments and change costing analysis for implementing a new application from the domain architecture. Certain expectations can be made when operating with a model at various normalization levels with respect to estimating effort and expense.

A database model can be designed in third normal form, but implemented in a denormalized fashion to balance integrity and performance. The same holds true for the domain model and its implementation. We are ultimately concerned with what level of normalization the actual domain architecture is in, not what the design may have intended it to be.

As with a normalized database, a normalized domain architecture should be able to have simple, powerful operations applied to it to create result sets. In a relational database, these results will yield a data set. In the context of a domain architecture, the result yields an application set. Just as the data returned from a relational database query may have to be structured for screen display or report printout, the application set returned from a domain architecture query typically needs to be formatted to become an executable.

4.4.1. First Normal Form

As shown in Table 4.2, establishing first normal form involves two tasks. First, we need to separate elements into affinity groups. We generalize this task by describing it in this format:

move < affinity-group-elements > into separate < affinity-group-domains >

Table 4.3 summarizes the implications of this task for each of the four domain model aspects. The second task is to define a primary key for each affinity group element. We need to identify those characteristics that make each element within each affinity group's extent (i.e. domain) unique. For a table, this represents one or more columns. For a class, this represents one or more member attributes. Establishing uniqueness is a relatively straightforward mechanism, usually established with an identity value or scoping within unique namespaces. In a single execution space, object identity is relatively straightforward. However, this can become more complicated with implementation strategies such as the *Flyweight* pattern [GHJV94], which manages the same object and references counts to it by other objects. Distribution also makes object identity more obscure, since parts of the instance may reside on different processors and some of those parts may be replicated, and thus be denormalized.

By their nature, use cases and policies are analysis constructs, components are design constructs, and transformations are implementation constructs. As just described, design and deployment issues present special problems for comprehending what uniqueness represents, as well as what separation into separate affinity groups represents. By definition, a use case represents a series of transactions between an actor and the system. Thus, first normal domain process form represents delineation of the system boundary, naming the actors, and naming the sequences of transactions that represent a distinct use of the system. A use case model typically provides this information, but the presence of a use case model does not assure first normal form unless roles have been abstracted for the actors.

Table 4.3. Affinity Groups for First Normal Forms

Aspect	Move…	Into separate…	That distinguish unique…	Represented as…
Relational	Attributes	Relations/tables	Entities	Tuples
OOAD	Attributes/methods	Classes/types	Entities	Objects
Process	Transactions	Use cases	Actors roles	Scenarios
Structure	Collaborations	Components	Operations	Functionality
Policy	Rules	Business Rule Statements	Policy Decision Parameters	Assertions & Derivations
Context	Deployments	Transformations	Applications	Modules

A business policy decision becomes manifest through business rule statements, which are then formalized through assertions and derivations. First normal domain policy form represents the clustering of such assertions and derivations into unique decision parameters. In this form, it is possible that an assertion or derivation can support multiple decisions. For example, a derivation may round all monetary calculations upward to the next penny. This derivation may represent a single policy or several policy decisions, such as for charging interest on outstanding balances or for calculating quantity discounts. In order to capture the correct intention, decision parameters are required to explicitly define the scope of each policy. Since such decision granularity is itself configurable for each application instance generated from the domain architecture, first normal domain policy form represents the clustering of such assertions and derivations into unique decision parameters at the highest possible level of commonality.

Instances of a component in first normal form represent alternative collaborative and algorithmic approaches to address the desired functionality. For example, the primary functionality may be to generate the grade point average for a particular student. One component may merely perform the calculation and return the value; another may make available a semester by semester breakdown with an additional breakout for classes that apply to the major. A query for determining grade point average will return both components, but one will likely be a better match. A similar example can be presented for describing how first normal form affects the context aspect. When creating an application, a number of transformations may yield a functionally equivalent executables capable of being deployed. One transformation may be able to convert a module for deployment on several operating system platforms, such as an OLE object on a client machine, a C++ object module for linking into an executable residing on an application server, or as procedural SQL code residing in a database engine.

4.4.2. Second Normal Form

Second normal form requires that the separation of attributes into separate entities when those attributes are only dependent on a part of the key. This takes the form of:

separate an < affinity-group-element > from the < affinity-group-domain >

if it does not depend fully on the identity of the < affinity-group-domain >

Since process and policy constructs are the primary forces for an analysis effort needed to expand the domain model, they become more vulnerable to major restructuring than the structurally oriented components and the context oriented transformations. Specifically, actors can be subsumed by the system. Also, derivations, once considered unique, can be found to share discovered generalization. It is in second normal form that interplay between process and policy first occurs.

Second normal form reduces a considerable amount of this potential volatility. For use cases, this means that redundancy caused by recurring transactions must be factored out into «uses» and «extends» use case stereotypes. To perform this separation, it is necessary to identify the key events that delimit each transaction. We thus separate a transaction from the use case if it can be triggered by an event that does not depend fully on the role that the actor is playing. If another actor can trigger the transaction, then the «uses» and «extends» use case stereotypes are required to convert the domain model into second normal process form.

Identification of these key events is represented in the policy context of the domain model as action controlling assertions that define the conditions for commencing a transaction. Furthermore, these assertions also dictate which of the scenarios for a given use case actually gets implemented. Consequently, second policy normal form requires a correspondence between use case transactions and action assertions though these key event definitions.

Second policy normal form also requires that we separate a rule from the business rule statement if it does not fully depend on the policy decision that forms the basis for the statement. In other words, if we parameterize the policy decision, the rule that gets invoked must be exclusively determined by that parameter. This implies that a rule can be specified independent of its actual use in any policy decision. This is particularly important for establishing key events. In general, policies map processes to components in order to establish a context for an application instance derived from the domain architecture. Therefore, policies represent the domain/extent/range of choices that can be made. As an analogy, consider each business rule as an automotive parts supplier, each derivation and assertion of a business rule as an individual part made by the supplier, and each application as an automobile. Consider each business policy decision as represented by an auto part list to build a component of the car, such as for the ignition system, cruise control, or headlight assembly. To implement this policy, the automobile manufacturer orders the specified parts from each supplier that has an item on the parts list. Similarly, we assemble our application from policies that call up the appropriate business rules resulting in a collection of formal assertions and derivations. Just as an automobile manufacturer can assemble a seemingly identical car by purchasing parts from alternate vendors, applications can be composed and generated from alternate business rules that implement policy decisions. In each case, performance, reliability, and quality may vary.

The purpose of this example is to illustrate the importance of second policy normal form in providing the glue that binds the other three aspects together. It allows us to consider domain application decisions that have not yet been addressed through process or structural constructs. As a result, meeting client requirements through fit assessment, exception handling, and change costing activities are easier to recognize and accurately predict.

Second structural normal form requires that we separate any collaboration of objects from the component if it is not required to support the complete set of public methods of the component. This is a straightforward definition that suggests that collaborations should be nested inside sub-components. There are exceptions to this rule that deal with objects that play multiple roles, but this deals with context rather than structure. Second structural normal form permits redundant classes that separate out different roles. Second context normal form, on the other hand, must make the distinction of separating a deployment from the transformation if it does not depend fully on the identity of the application. We provided an example of how one transformation may be able to convert a module for deployment on several operating system platforms to illustrate first context normal form. To move that same example into second normal form, the deployable modules must separate out the language aspect (i.e. C++, SQL code) and the platform aspect (i.e. client, application server, database engine). Thus we are capable of capturing deployment options with respect to language and environment that can be made available in latent policy decisions.

4.4.3. Third Normal Form

Third normal form requires that the separation of attributes into separate entities that exhibit transitive dependencies. This takes the form of:

separate an <affinity-group-element> from the <affinity-group-domain>

if it is transitively dependent upon the <affinity-group-domain>

Through use case refinement, we identify nested transactions based on speech acts in which the performer becomes the customer in the decomposition. This represents a change in the system boundary. Such subsystems are intended to align with the structural aspects of the domain model manifested through components. Thus third normal form represents unification of the process and structural aspects of the domain model through representations such as sequence diagrams, where use case steps are represented as message invocations and components are represented as the message targets. Use case refinements demonstrate transitive dependency by exposing details previously unknown. This could equate to the sales invoice for an automobile that includes the basic information of the sale, including total retail price, but refinement reveals itemization of the options and their cost. Collaboration refinement provides the same separation for the structural aspect of the domain model normalization process. Since third normal form requires a one-to-one correspondence between the use case scenarios and collaborations within and among components, it is likely that the selection of components will be accompanied by attached use cases. This presents its own unique problems that must be resolved by the policy aspect of the domain model. Policy may dictate that unneeded refinement is removed. It may, on the other hand stipulate that the additional functional capabilities be included as influencing conditions for application deployment, but not required. This gives rise to the notion of anti-scenarios, unwarranted scenarios, and under-specified scenarios. In the case of anti-scenarios, policy stipulates behavior that the system must not demonstrate. Unwarranted scenarios describes functionality that exists, but is not required, therefore no assurance as to reliability, quality, or future availability are made. Under-specified scenarios occur when structural and process normalizations identify required policy decision parameters that have not been specified.

Third policy normal form carries the symmetry of a unified structural and process model to completion by associating a policy decision parameter at each component boundary. Since process and structural refinements are by definition transitively dependent, associated policy decision parameters that refine previous, larger grained parameters are likewise transitively dependent. Thus, in order to place the domain model in third normal form, the set of parameters must be separated along the same sub-system boundaries that are represented as actor/system boundaries in the process view and as inter-component communication in the structural view. Once such refinement has been made, pre-defined architectural templates can be specified by attaching default values to parameters. These mega-parameters may still exhibit transitive dependencies since the degrees of freedom for the parameter value may be restricted by other parameter elections made.

When considering the implications of third policy normal form, we must keep in mind that we are not assured a minimally configurable implementation. By this we mean that no assurances can be made that a fit assessment will reveal this most efficient implementation strategy. All that can be guaranteed is that the domain model will demonstrate certain properties that make it easier to identify a “satisificing” set of policy decision parameters coupled with domain model operations needed to grow the domain model to a level necessary to meet application implementation requirements.

Third context normal form requires that unless a deployable module is directly associated with another module, then any deployment specific code must be removed. For example, in a client-server architecture, a middle layer may be required to connect the client to a SQL database engine, such as through ODBC drivers, or a native database engine driver encapsulated in an OLE object. In third context normal form, the client cannot contain any engine specific syntax, such as pass through queries.

4.4.4. Additional Normal Forms

Fourth and Fifth Policy Normal form are patterned after the corresponding relational normal forms. Fourth relational normal form deals with elimination of redundancy that is caused by having multiple 1:N relationships in the same table. For example, a table consisting of {person, food, wine} should be split into two separate tables consisting of {person, food} and {person, wine}. This requires understanding the meaning of the relation. Fifth relational normal form is established when a relation cannot be reconstructed by a join operation of two or more smaller relations. This occurs only when there is a symmetric constraint on the data values. Both of these situations are most likely to occur when dealing with policy decision parameters. When the mega-parameter exhibits a symmetric constraint imposed by sets of parameters, resolving to fifth normal form removes any remain redundancy imposed by such constraints. Policy is the only aspect that is of interest with these higher relational normal forms in our treatment of domain normalization. Therefore, symmetric constraints are addressed in the fourth normal policy (rule) form. Table 4.4 identifies how each of the aspects relates to arrive at a particular domain normal form. This table does not preclude the relaxation of normal form in one aspect that is compensated for in another aspect.

Table 4.4. Domain Normal Forms

Form	Structure	Process	Rules	Context
1DNF	1SNF	1PNF	1RNF	1CNF
2DNF	2SNF	2PNF	2RNF	2CNF
3DNF	3SNF	3PNF	3RNF	3CNF
4DNF	3SNF	3PNF	4RNF	3CNF

4.5. Discussion

This chapter introduced the concept of domain normalization that integrates structural, process, policy, and contextual aspects of a business domain-specific architecture. The motivation behind this effort was to provide a foundation for managing the evolution for such a domain architecture. The structural aspects of a domain architecture are the best understood. Considerable literature from the domain engineering and object-oriented analysis communities is available. At least one other attempt [Ambl97] has been made to apply normalization to a class structure. Accordingly, this aspect will not further elaborated upon, except in the context of the other three.

The next three chapters of this thesis will focus on these remaining three aspects. In [Chen95], a similar approach attempted to integrate process and policy aspect of a multi-model system architecture [Su91]. However, Chen's work was primarily restricted to extensional operations. In this thesis, the primary focus is on the intensional operations that affect the domain architecture itself. In chapter five, adaptive use cases address the needs for process normalization. That chapter will present model elements specifically designed for applying *expansion* and *integration* operations that factor common use case actions. Chapter six will introduce parameterized business rule patterns that address the needs for policy normalization. Examples are presented that will demonstrate the *refinement* and *extension* operations to improve policy (rule) normalization. Chapter seven then discusses transformations for use case formalisms that deal with the contextual aspects of normalization as well as demonstrate how to apply adaptive use cases. That chapter also extends the concept of parameterization by demonstrating how the deployment of an application can be facilitated through application of the pattern language. Finally, fit assessment and change costing analysis for implementing a new application from the domain architecture are elaborated upon in chapter seven. These two component models quantify each of the domain normalization operations through metrics that can calculated and then used to gauge the maturity of the domain architecture through its defined attributes.

CHAPTER V

ADAPTIVE USE CASES

This chapter formally defines an adaptive use case meta-model. The structural and dynamic relationships of each model element are presented using Unified Modeling Language notation (UML) and semantics. Specifically, this chapter presents definitions for use cases and related concepts from the UML specification documents. This is followed by a discussion of the limitations and potential pitfalls of building use case models using UML semantics. Then, each of the conceptual elements that form the foundation for adaptive use cases are discussed with respect to how they relate to the UML specification. These components are the Workflow Application Programming Interfaces from the Workflow Management Coalition's Workflow Reference Model [Holl94], speech act semantics based on the ActionWorkflow model [MWFF92], and adaptive modeling elements based on Basset's frame technology [Bass96]. The Workflow Reference Model is presented through versions of the Process Definition and Client Application interfaces that are developed using UML structural notation. Speech act semantics are placed in the context of a use case refinement level model that is introduced in order to relate various stakeholders to the view focus of an abstraction level. Adaptive frames technology is applied to the UML use case model to develop the basis for defining the two major constructs of the adaptive use case meta-model: the *Scenario* model element and the «adapts»[2] stereotype relationship between *Scenario*.

Having described each of the major components that are integrated into the conceptual framework of the adaptive use case meta-model, each of the adaptive use case model elements is then described in terms of structural and behavioral semantics. Business rule constructs, which are an integral part of the adaptive use case meta-model are also introduced as a lead in to the presentation of business rule models, patterns, and templates in the next chapter.

5.1. Unified Modeling Language Use Case Definitions

There has been a great deal of ambiguity regarding the definition of a use case. Differences of opinion about the scope and plurality of use cases are common [Cock97]. The UML specification attempts to establish a clear understanding as to exactly what the use case construct represents. In the course of

[2] Following UML conventions, names of stereotypes are delimited by guillemets and begin with lowercase (e.g., «type»).

defining a use case, two additional important terms emerge. These two terms are *scenario* and *generalization stereotype*. Each of these three concepts is fully defined for every context in which the term is used within the UML specification. These definitions shape what constraints the UML places on potential use case formalisms that may be defined through its extension mechanisms.

5.1.1. Use Case

The Unified Modeling Language UML specification version 1.1 [UML97] provides several perspectives for its definition of a use case:

- <u>Glossary</u>: "The specification of a sequence of actions, including variants, that a system (or other entity) can perform."
- <u>Notation Guide</u>: "A use case is a coherent unit of functionality provided by a system or class as manifested by sequences of messages exchanged among the system and one or more outside interactors (called *actors*) together with actions performed by the system."
- <u>Abstract Syntax General Presentation</u>: "The *use case* construct is used to define the behavior of a system or other semantic entity without revealing the entity's internal structure. Each use case specifies a sequence of actions, including variants, that the entity can perform, interacting with actors of the entity."
- <u>Abstract Syntax Metaclass Definition</u>: "In the metamodel *UseCase* is a subclass of *Classifier*, containing a set of *Operations* and *Attributes* specifying the sequences of actions performed by an instance of the *UseCase*. The actions include changes of the state and communications with the environment of the *UseCase*."
- <u>Dynamic Semantics</u>: "Each use case specifies a service the entity provides to its users, i.e. a specific way of using the entity. It specifies a complete sequence initiated by a user; i.e. the interactions between the users and the entity as well as the responses performed by the entity, as they are perceived from the outside, are specified. A use case also includes possible variants of this sequence, e.g. alternative sequences, exceptional behavior, error handling etc. …Each (use case) individually describes a complete usage of the entity."

- Notes: "A pragmatic rule of use when defining use cases is that each use case should yield some kind of observable result of value to (at least) one of its actors. This ensures that the use cases are complete specifications and not just fragments."

5.1.2. Scenario

The UML defines a scenario only within the context of other modeling elements. This can lead to some confusion, particularly with the circular nature of the references to these other modeling elements.

- Abstract Syntax: "An explicitly described *UseCaseInstance* is called a *scenario*" and a use case instance is defined as "the performance of a sequence of actions being specified in a *use case*".
- Glossary: "A specific sequence of actions that illustrates behaviors. A scenario may be used to illustrate an interaction." An interaction is defined as 'the specification of how messages are sent between instances to perform a specific task … in the context of a collaboration.' A collaboration is defined as "the specification of how a classifier, such as a use case or operation, is realized by a set of classifiers and associations playing specific roles, used in a specific way. The collaboration defines an interaction."

5.1.3. Generalization Stereotypes -- «extends» and «uses»

The UML specification defines two generalization stereotypes as follows:

- Uses: "Commonalities between use cases are expressed with *uses* relationships. The relationship means that the sequence of behavior described in a used use case is included in the sequence of another use case. The latter use case may introduce new pieces of behavior anywhere in the sequence as long as it does not change the ordering of the original sequence. Moreover, if a use case has several uses relationships, its sequence will be the result of interleaving the used sequences together with new pieces of behavior. How these parts are combined to form the new sequence is defined in the using use case."
- Extends: "specifies that the contents of the extending use case may be added to the related use case. It not only specifies where the contents should be added (*extensionPoint*), but also if it only should be added if a specified *condition* (BooleanExpression). When an instance of the related use case reaches the extension point and the condition is fulfilled, the instance continues according to a sequence that is the result of extending the original sequence with the extending sequence at this point. It is required

that the ordering of the parts of the extending use case must be fulfilled if its parts are inserted at different places."

5.2. Limitations of UML Modeling Constructs

The inclusion of the «extends» and «uses» stereotypes has been inherited from the Jacobson, *et al.* Object-Oriented Software Engineering (OOSE) method [JCJO92]. Use cases under OOSE were not subject to the same level of precision that the UML semantics places upon them. In a less structured context, «extends» and «uses» conveyed intentions through concise representation at the expense of precision. This tradeoff of expressiveness for exactness still works well for higher level analysis between domain experts and developers. However, once progressing into detailed analysis and design, these two stereotypes are less effective. Moreover, the conversion of the «uses» stereotype from a dependency relationship in UML 1.0 to the current manifestation as a generalization relationship in UML 1.1 highlights the struggle to accommodate these stereotypes in the UML meta-model.

5.2.1. Use Cases May Describe Incomplete Usage of the System

Figure 6.12 illustrates a use case system diagram and a use case relationships diagram using the example provided in the UML Notation Guide. Several problems can be observed from these examples. First, the *Telephone Catalog* system probably does not involve direct interaction with the customer. From the *customer's* perspective, the system includes the *salesperson* and *supervisor*, since he otherwise has no direct knowledge of the system. Second, the use case system diagram does not contain any reference to the *Supply Customer Data, Order Product, Arrange Payment,* or *Request Catalog* use cases. This omission is intentional since they do not represent a complete usage of the entity. However, these use cases *do* yield some kind of observable result of value to at least one of its actors. A customer could not order product without supplying customer data. Even if the customer had an account, a name or identification number would be required.

The essence of this difficulty with «extends» and «uses» lies in the contradictory definitions of a use case and not in the definition of relationships between use cases. From the perspective of a single actor, it is debatable as to whether a "used" or "extending" use case constitutes a complete usage of the system. Regardless, there is nothing to prevent this from occurring. However, when expanding the number of conceivable actors, it becomes more plausible for a use case that is considered to be a piece of a complete usage for one actor to be considered the complete usage for a different actor. This presents a dilemma for the domain modeler. An evolving use case model that anticipates refinements to include additional actors would not be well formed until such refinement occurred. This forces the modeler to either defer creation of the "used" use cases until the secondary actors are identified or initially include more detail than desired.

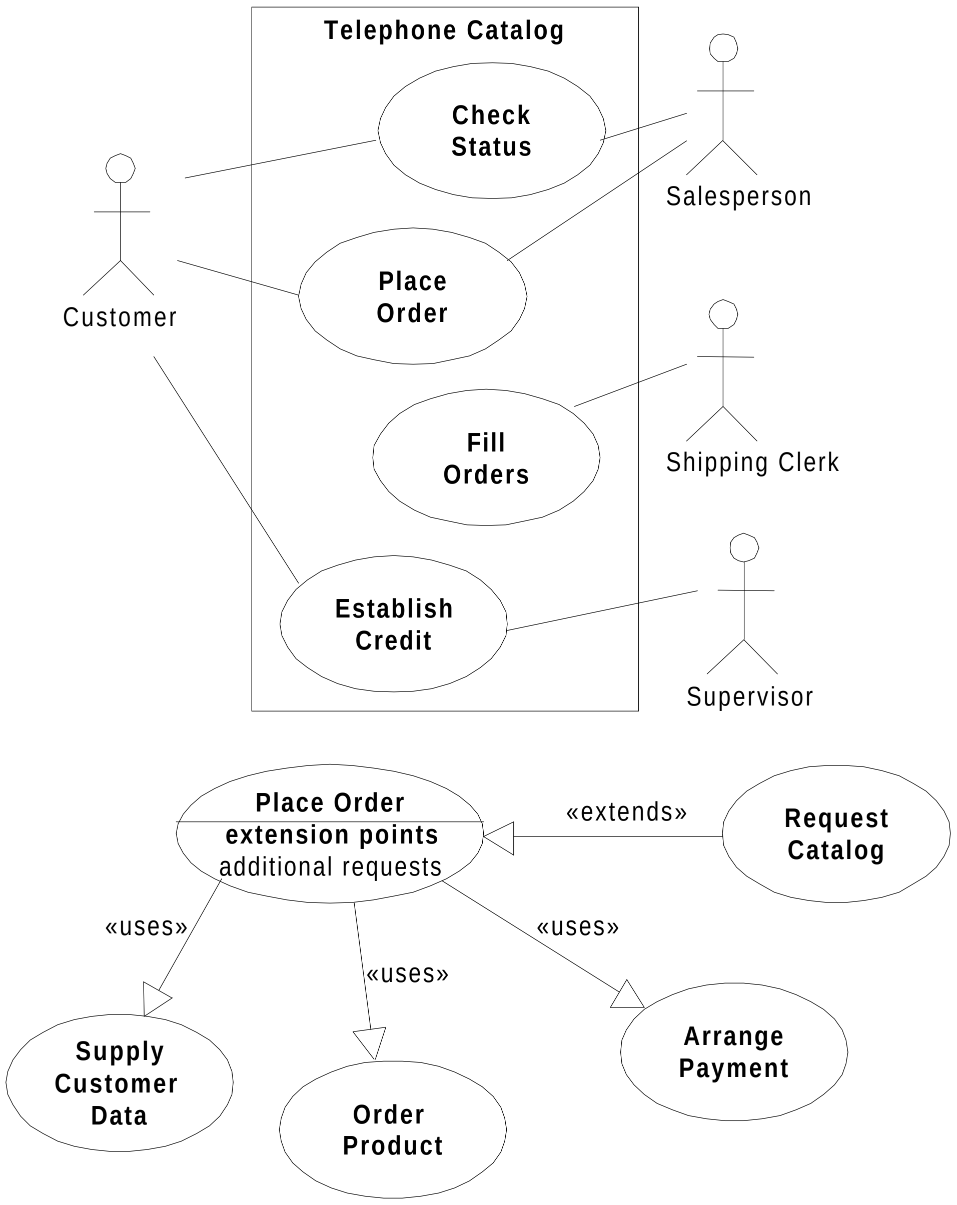


Figure 5.1. UML Use Case Notation

5.2.2. Use Cases May Incorporate Incomplete Reuse of Another Use Case.

Most of the commonality within a use case model does not occur between use cases. Instead, commonality most often occurs as a factored sequence of actions from one or more scenarios within a single use case. If a use case is considered a collection of scenarios [Cock97], [KPW97], [RD97], then a "using" use case must consider all of the scenarios of the "used" use case. Complete reuse of a "used" use case implies that the complete set of alternative behaviors in the "used" use case is possible at some point during an interaction between the actor and the system. However, within the context of the "using" use case, the full range of behavior may not be possible or desired. Instead, restrictions on the behavioral degrees of freedom within a "used" use case are only discernable when viewed from the perspective of each individual scenario refinement. This problem occurs because the complexity of the «uses» generalization is not specified at the same level of abstraction that the relationship is defined. The relationship can only be fully specified as the aggregation of all nested scenario relationships contained within the two use cases.

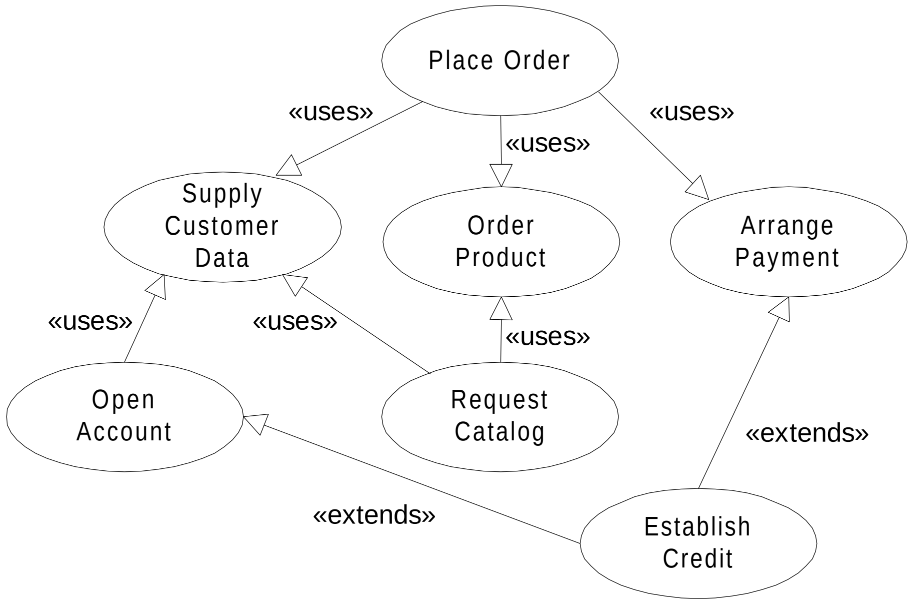


Figure 5.2. UML «extends» and «uses» Use Case Generalization Stereotypes

To illustrate the problem of incomplete reuse of a "used" use case, the *Telephone Catalog* set of use cases from Figure 5.1 is expanded to include electronic catalog orders via web pages. One new use case, *Open Account*, is added and the dynamic semantics of the *Request Catalog* and *Establish Credit* use cases are modified, as shown in Figure 5.2. Many of these changes are intended for later illustrations. For now, consider the *Request Catalog* use case. The changed semantics for the *Request Catalog* use case in our example resulted from an expansion of functionality that permitted a prospective customer to receive a catalog. Here, making a catalog request is considered the same as *Place Order*, except that there is no charge, so *Arrange Payment* is unnecessary. However, the actor may be a prospective customer rather than an established customer. For a prospective customer, certain pieces of information may be unneeded, such as a billing address. Therefore, portions of the "used" *Supply Customer Data* use case should not be invoked. Attempting to resolve this problem by factoring out functionality into another use case only exacerbates the incomplete usage of the system problem.

Similar problems occur for an «extends» relationship. Consider a customer that is ordering electronically through the web pages rather than by phone. In this context, the supervisor will be unavailable to immediately approve credit in the customer desired to complete the invocation of *Arrange Payment* through placing the order on account through the *Establish Credit* "extending" use case. Although the «extends» permits conditional tests before invoking the "extending" use case, forcing each invocation of the "extended" use case to perform this test lead to a combinatorial explosion of tests. Such tests should be eliminated through pruning away scenario paths that may no longer occur once the context of the use case has been established at its initiation.

5.2.3. Use Cases May Require Shared Conditional Extending Behavior

The UML specification states that commonalities between use cases are expressed with «uses» relationships. However, only «extends» relationships have conditions attached to them. Referring to the *Establish Credit* use case in Figure 5.2 again, this use case can occur as extensions to both the *Arrange Payment* and *Open Account* use case. Any modeling solution requires some contrivance. For example, a «uses» relationship could be established with a dummy placeholder use case that is then extended by *Establish Credit*. Alternatively, *Arrange Payment* could establish an «extends» relationship with *Place Order* that is conditional upon its not being invoked by the *Supply Customer Data* use case, however this further requires that the relationship between *Open Account* and *Supply Customer Data* be changed to an «extends» relationship.

The simplest alternative is to convert the relationships involving *Establish Credit* to «uses». However, if this association was changed, as suggested in the previous paragraph, then "complete usage" is violated, since *Establish Credit* should only be permitted under certain scenarios of *Arrange Payment*. An expedient solution would be to add conditions to the «uses» relationship and eliminate the distinction by removing the «extends» relationship. However, semantic meaning for informal use cases that are useful in early analysis of a problem domain would be compromised.

5.2.4. Interleaved Use Cases Are Indistinguishable From a Single Use Case

It is quite possible that only the actor may be able to determine whether his interactions with the system comprises a single use case or a collection of two or more use cases that may be temporally interleaved, but otherwise independent of each other. We have shown through examples that the *Request Catalog* and *Establish Credit* use cases can appear either as stand alone use cases or as an integrated part of a larger use case that reflects the goal of the actor. Suppose that the customer intended not to place an order unless a line of credit was extended. It is evident with such a goal, that the *Establish Credit* use case is an essential part of the actor's intended usage (assuming that he does not yet have a credit line). However, if a salesperson is taking an order by phone and merely suggests that the customer obtain a line of credit or offers to send out the most recent catalog, the actor's initial goal of placing an order has not changed. In such a context, *Establish Credit* and *Request Catalog* may be considered separate, but temporally interleaved use cases with the *Place Order* use case.

Such a determination often goes beyond simple *a priori* consideration of the actor's goal [Cock97] or of the scope of a scenario [Beri97]. The unfolding of the use case dialog alters an actor's objectives as well as the length and course of the dialog's sequence actions. For example, the salesperson informs the customer of a special product offer that alters the customer's goals such that the order will be placed in spite of being turned down for credit.

The system may not care whether the actor considers the series of interactions one or more use cases. However, if the actor represents a component, which itself is composed of cooperating components, the use cases for the refinement of the component may need to make that distinction. In other words, goals and beliefs of the component playing the role of an actor will affect its behavior. Thus under UML semantics, the system must be modeled with use cases that view the actor's intended usage either way.

It is also possible that interleaving of use cases may not become apparent unless a different abstraction level is considered. For example, two instances of the same actor, such as two employees from the same company, may both be playing the actor role of customer. Two separate orders may be placed on the same day and then treated as a single shipment by the shipping clerk. The shipping clerk is not concerned with the fact that two use case instances were responsible for his use case instance. Receipt of both orders as a single shipment may represent the same step of a *Restock Supplies* use case. Allocation of responsibilities among the use cases that represent each component of a subsystem may have inter-relationships that require a distinction to be made between independent use case instances and inter-dependent ones. Essentially this means that a superordinate use case is partitioned into segments that are fulfilled by the use cases of the components that are contained in the superordinate component [JGJ97]. The design model for the superordinate use case must be considered. The UML specification does not adequately express semantics for the mapping of the superordinate use case to the subordinate ones.

5.2.5. Actor Class Hierarchies Induce Homomorphism

Within the context of a use case model, homomorphism represents the parallel mapping of an actor inheritance hierarchy and a use case inheritance hierarchy of «uses» relationships. Continuing with the *Telephone Catalog* extended example, Figure 5.3 shows an actor inheritance hierarchy and one use case. Since *Place Order* is associated with several other use cases, each of the "used" use cases may also have to be subtyped to match the actor subtypes.

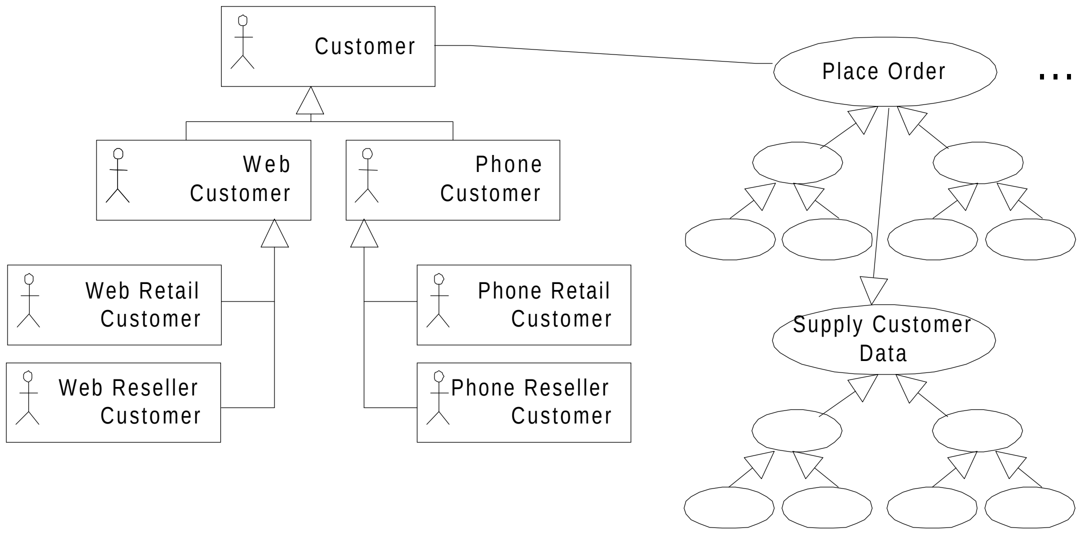


Figure 5.3. Use Case Homomorphism

Since each combination of actor subtypes may involve different parameter types associated with the messages it sends to the system, the behavior may be slightly different in each context. For example, a reseller customer may need to furnish a reseller identification number document as part of the *Supply Customer Data* use case. This may be further refined by the mode of contact: the phone reseller customer may be able to mail or fax the document; the web reseller customer may be restricted to furnishing a binary file. The retail customer may have its sales tax computed by the *Place Order* use case depending on the shipping address. To accurately model the domain using UML semantics, the use case modeler runs the risk of getting lost in an explosion use case permutations. Refraining from such elaboration leads loss of precision or carrying around excess baggage in each use case in the form of conditional information based on the actor subtype.

5.2.6. Avoiding UML Modeling Difficulties with «extends» and «uses»

A use case modeler needs to be careful with how UML model elements are applied when constructing domain models. The following guidelines are presented for avoiding potential pitfalls when confronting limitations of UML use case semantics:

- Don’t put the generalization arrow in the wrong direction. The generalization arrow for «extends» intuitively goes in the wrong direction. “Extends” is more appropriate as a role name in an association than the name of a generalization.
- Don’t use «uses» for conditional relationships between use cases. Only «extends» has conditions associated with the relationship. This implies that all “used” use cases are unconditional. Given the collection of scenarios that a use case represents, this is an unreasonable expectation for common behavior to be used in each scenario.
- Don’t forget to add an extension point to the extended use case when defining an «extends» use case relationship. By associating extension points with the extended use case, extending that use case either requires a change for adding the new extension point or *a priori* knowledge of all possible points where the use case could ever be extended.
- Don’t create use case fragments. A “used” use case is often not a complete specification. Naming of common behavior that has been factored out is often mistaken for a complete use case. Forcing factored behavior to be a complete specification is too limiting.

- Don't try to name all scenarios for complex use cases through use case sub-typing. Combinatorial explosion of scenario permutations often prevents explicitly describing (or naming) each scenario. Although a use case may be considered as a collection of all possible scenarios between an actor and the system for a given purpose, UML semantics provide no mechanism to enumerate all possible scenarios.
- Don't reveal internal structure in a use case. There is a temptation for use cases to reveal internal structure simply because they would be awfully boring otherwise, i.e. nothing more than a protocol description. Impact on the environment and perceivable changes to internal state of the system should drive the chunking of actions into a single use case action step.
- Don't select different use case representations for the same entity. Use cases may have a wide variety of representations, such as narratives, collaboration diagrams, interaction diagrams, state machines, and activity diagrams. Unlike *Class* model elements which have widely varying attributes, attaching different attributes to each *Use Case* model element runs contrary to the spirit of standardization of the UML notation and semantics.

5.3. Workflow Reference Model Workflow Definitions

The Workflow Management Coalition's (WfMC) has created a Workflow Reference Model (WRM) that provides several definitions related to a business process [Holl94]:

- Workflow: The computerized facilitation or automation of a business process, in whole or part.
- Workflow Management System: A system that completely defines, manages and executes "workflows" through the execution of software whose order of execution is driven by a computer representation of the workflow logic.
- Process Definition: The computerized representation of a process that includes the manual definition and workflow definition.
- Workflow Control Data: Internal data that is managed by the workflow management system and/or workflow engine.
- Workflow Relevant Data: Data that is used by a workflow management system to determine the state transition of a workflow process instance.

- Workflow Application Data: Data that is application specific and not accessible by the workflow management system.

5.3.1. Process Definition Model in UML Notation

The WRM defines a set of components that communicate via five interface specifications for the exchange of information. The Workflow Application Programming Interfaces Interface 1 (WAPI-1) is a process definition read/write interface that provides a formal specification to access workflow definitions [WfMC95]. Several standard entity classes and associated attributes are defined. These are shown in Figure 2-1 using UML notation with some allowances for naming conventions. Three levels of compliance are specified. *Level A* compliance simply defines access to the workflow definitions. The format, class, and attribute structures are part of *Level C*. *Level B*, which actually deals with the workflow definition structure, has not been specified. However the format for any workflow definition must be provided in terms of the *ClassDefinition* and *AttributeDefinition* structures. As such, the WfMC provides a method to directly map the UML meta-classes into its workflow definitions.

The WAPI-1 definition enables an application to query the workflow engine as both how the workflow definition is structured and the workflow definition itself. In the context of the UML meta-model, this relates to the description of the *UseCase* meta-model element and an instance of *UseCase*, e.g. *Place Order*. In order for this interface to work within the UML specifications, there needs to be a more formal, structured definition of a use case meta-model element. Therefore, each type of use case representation must be mapped to a specific *Format* definition shown in Figure 5.4. State machines and activity models have well defined semantics within the UML. Although it is a relatively straightforward process to present such models tailored specifically for use case representation, there is still no formal representation for structured text.

Converting the structured text use case into a WAPI-1 compatible format can be accomplished through one of three basic approaches. First, the use case modeler may manually generate a WAPI-1 compatible format based on the structured text use case. Second, the use case modeler may manually convert the structured text use case into a more formal UML representation, such as an activity model or state machine model, and then automatically transform the more formal representation into a WAPI-1 compatible format. Finally, the use case modeler may adopt a more formal structured text use case representation by defining more formal use case model elements.

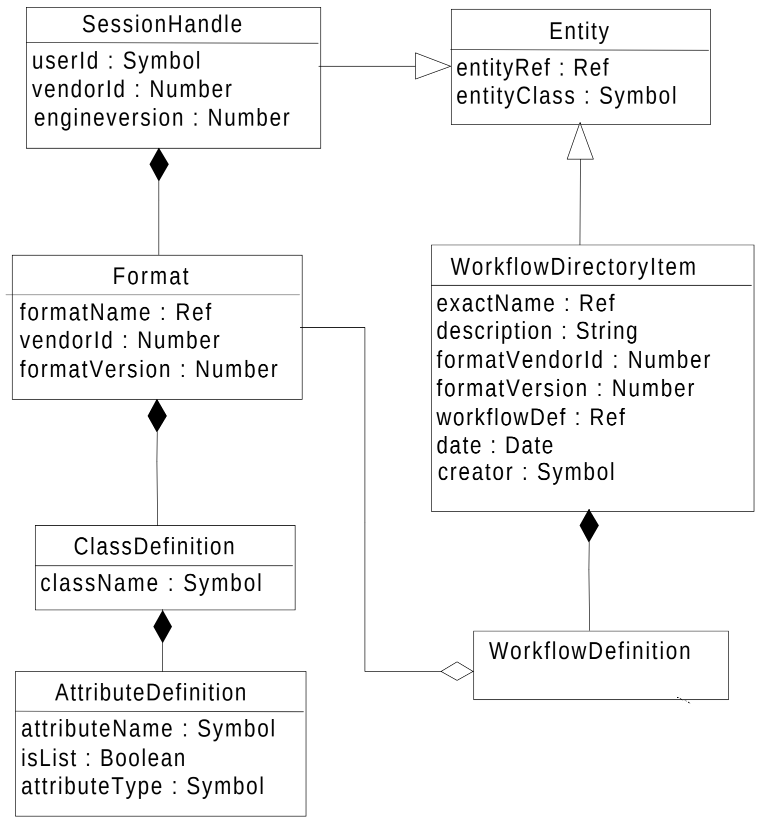


Figure 5.4. WfMC Process Definition Interface 1 Specification

Our approach for accommodating the WAPI-1 specification adopts this last approach, i.e. by defining new UML modeling constructs through use of the UML *Stereotype*, *Constraint*, and *TaggedValue* extension mechanisms. Once mapping to structured text use cases is defined through the UML extensions, mappings to use case refinements and realizations are possible. Extending these mappings to other meta-models, such as the WfMC WAPI-1 Process Definition meta-model or a business rule meta-model becomes a one time process. The use case modeler will no longer need to craft each new use case by hand when representations in alternative meta-models are desired. Furthermore, use case reification is accomplished. Since the workflow engine captures the process and activity states, the actual execution history of a use case can be captured through the audit facilities provided by the workflow engine. This reification is found as a recurring architectural pattern in applications and development environments [Beed97], [Jans95].

5.3.2. Client Application Interface Model in UML Notation

The WfMC WAPI-2 Workflow Client Application Interface Specification was created for the purpose of providing a consistent method of access to the process control and status functions of a workflow engine. Figure 5.5 depicts the WAPI-2 meta-model elements in UML notation. The elements from the WAPI-2 specification are combined with the Common Object Model OLE and CORBA IDL Bindings to form a more object oriented graphical depiction of this interface.

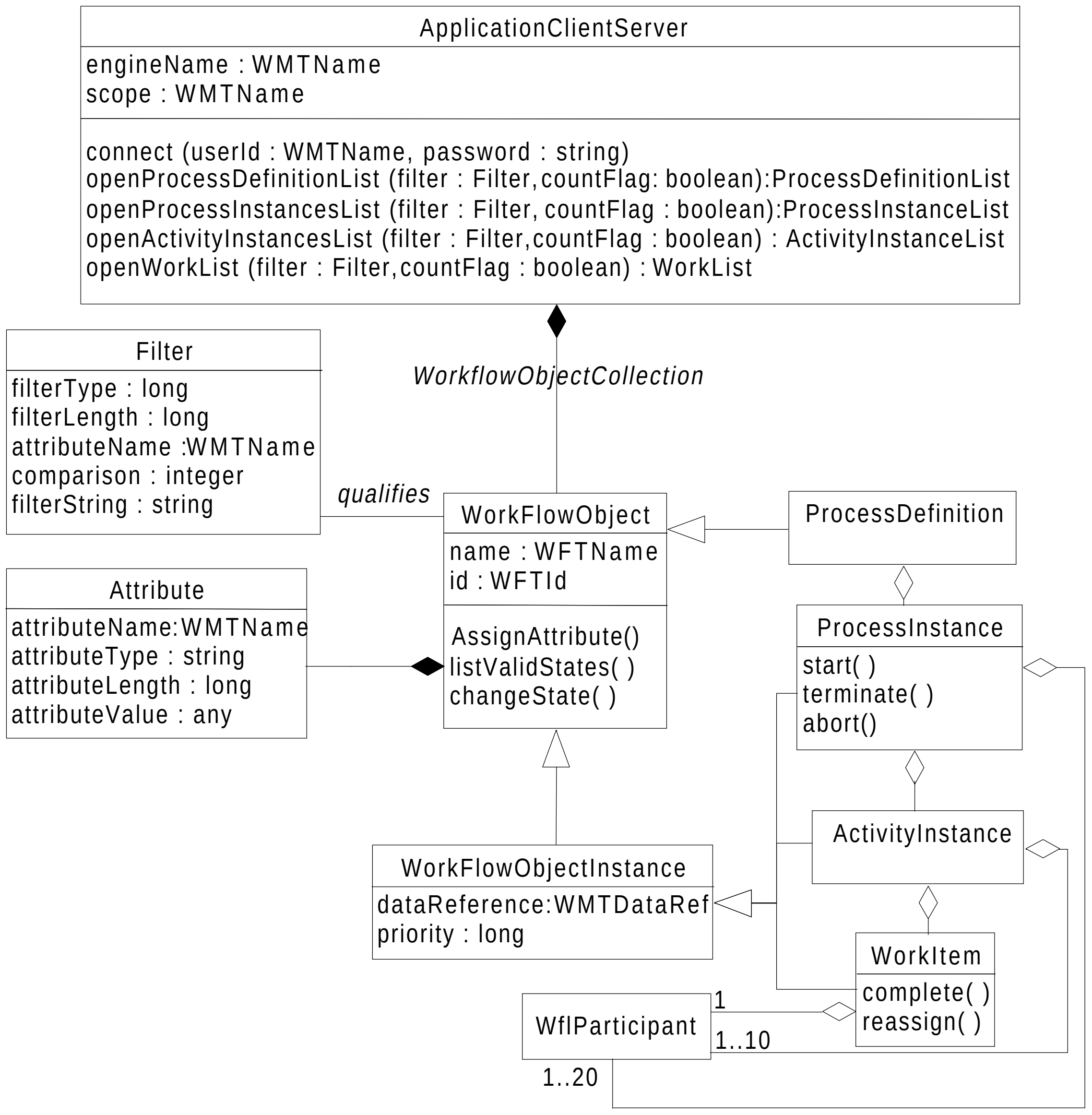


Figure 5.5. WfMC Client Application Interface 2 Specification in UML Notation Format

Several liberties were taken in the model. First, the *WorkflowObjectInstance* model element was created to factor out commonalities among the *ProcessInstance*, *ActivityInstance*, and *WorkItem* model entities. Second, The *WflParticipants* entity was added to the model to illustrate the relationship of use case actors to the elements of the *ProcessDefinition* construct. Third, inconsequential details were suppressed, such as iterator classes, container classes, and accessor methods.

This depiction of the Client Application Interface meta-model enables a comparison with use case meta-model elements to be made. Many similarities exist between a WfMC *WflParticipant*, *ProcessDefinition*, *ProcessInstance*, and *ActivityInstance* and a UML *Actor*, *UseCase*, *UseCaseInstance*, and an enumerated step of a *UseCaseInstance* respectively. In many contexts, these model elements may be considered equivalent. However, in other contexts distinctions must be made. In the WfMC model, a *WftParticipant* corresponds to any non-automated UML *Actor* of a *UseCase*. Within a single *ProcessDefinition* all automated systems must be considered inside a single system boundary. Thus, the WfMC process may represent a larger grained aggregation of several UML *UseCases*. Conversely, the workflow engine must also be included inside the system boundary. Therefore, a UML *UseCase* containing a meta-level WfMC *ProcessDefinition* is needed to describe the creation of the process *ProcessInstance* from *ProcessDefinition* and the update of workflow relevant data, such as status and assignment of roles to actual participants.

5.4. Speech Acts

Integrating the concept of speech acts into use case modeling places additional constraints upon use cases as defined in the UML. A speech act, as represented in the ActionWorkflow model, consists of a dialog between a *Customer* and a *Performer* [MWFF92]. Accordingly, representation of the interactions with secondary actors is suppressed in use case models. Sequence diagrams also follow this structure of dialogs. These secondary actors are only revealed indirectly through realization of the use case as collaborations of subsystems and classes. In order accommodate speech acts into the UML definition of use cases, three key points must be reconciled:

1. A use case describes a complete usage of the system.
2. A use case instance can communicate with more than one actor.
3. A use case does not reveal the internal structure of the system.

The ActionWorkflow model consists of four phases: preparation, negotiation, performance, and acceptance. The basic speech act model can be overlaid on the ActionWorkflow model to reveal alternative and exceptional courses for the dialog, such as counters, rejections, withdrawals, and reneges.

5.4.1. Use Case Refinement Level Model

Since the definition for a use case includes specification of all responses performed by the system that are perceived from the outside, it is possible that more than one workflow dialog will be included in the use case. The use case of one actor may require that the system initiate interactions between the system and other actors in order to respond completely. The disclosure of these secondary actors, which are of no real interest to the primary actor, comes as a result of the refinement level.

Use cases, as defined by the UML, are system-oriented. The focus is a single entity that represents the system along with other entities outside the system that are modeled as actors. Speech act semantics based on the ActionWorkflow pattern shift the focus to an actor-oriented view. Only a single actor is considered and the system scope can vary to include those entities modeled as actors in the system-oriented view. Both views are relevant. Refinement of a use case may occur all the way down to the *Class* level. Interest in the visibility of entities outside the system at any given abstraction level depends on the stakeholder.

This thesis has developed a model, shown in Figure 5.6, that illustrates the correspondence between stakeholders and view focus based on the use case refinement level. Each box in this model represents a system boundary at a certain level of use case refinement, which each enclosed box representing a finer level of abstraction. At the lowest level of abstration, the implementer is concerned with use cases for classes and how they cooperate to fulfill superordinate use cases. The designer is more concerned with a higher level of abstraction that encapsulates the sets of collaboration patterns used to fulfill the use cases at the framework level of abstraction. At this level, the designer is concerned with the collaboration of frameworks to fulfill the use cases of components. Progressing to higher levels of abstraction, the architect is concerned with component collaborations to fulfill the use cases at the system-wide level. The customer of the system brings the model back the the level that corresponds the the top-most level defined in the UML. This level excludes secondary actors since they represent functionality outside to scope of the system as specified in the requriements model. The highest level corresponds to a speech act based use case dialog which only considers one actor at a time.

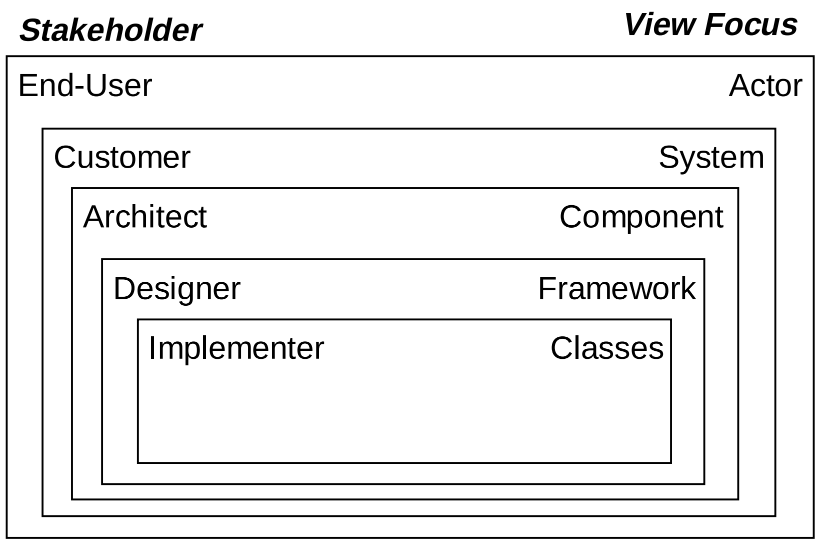


Figure 5.6. Stakeholder/View Focus Use Case Refinement Level Model

The end user at one level is merely a system component at a higher level. Therefore, an actor-oriented view represents different things to different stakeholders. Relevant scenarios from higher level use cases can be assembled to allow the stakeholder to establish a context for the system entity at his level of abstraction. Each of the actor-system dialogs that are revealed map direcly to ActionWorkflow patterns. When dealing with business processes that are only partially automated, the ability to open up the black-box behavior of the system boundary becomes a valuable mechanism for establishing contexts and responsibilities in defining workflows and processes. Moreover, as components, frameworks, and classes are reused to build an application system, this glass-box view facilitates the identification of an appropriate level of refinement.

Another advantage of placing use cases into a larger context is that it reveals how collaborations between actors of the lower-level black-box systems can be structured. From such a perspective, opportunities may surface to consolidate behavior that might not be apprarent if use cases are only considered as refined decompositions of superordinate use cases for higher level system boundaries. This synergistic combination is refered to as the *Shanley Principle* [Jack94].

Use cases were originally intended to depict how humans enter and interact with object oriented systems [JCJO92]. With the addition of workflow and automated agents, this concept must be expanded to embrace how humans or organization interact and how subsystems interact. Therefore, several scenarios of interaction are possible, as shown in Table 5.1.

Table 5.1. Customer-Performer Interactions between Human and System

Customer	Performer	Representation
Human	System	Traditional use case
Human	Human	Traditional workflow
System	System	Agent based workflow/use case leveling
System	Human	Automated workflow

5.4.2. Example

The use case shown in Example 5.1 is used to illustrate how speech acts are used to add additional constraints to a use case with respect to system boundary. This example is provided in narrative format. In addition to the use case narrative, two business rules and two functional variations that can be added to the base use case description are described. A structure version will be presented later as part of Figure 5.8.

The manual intervention of the department administrative assistant may not be discovered until the alternative branches have been specified in the use case. How this use case is refined depends on where the system boundaries are placed. Applying the use case refinement level model from Figure 5.6, the system boundary may be placed at any one of these boundaries:

1. Component oriented view – the automated portion of the student registration system. Two separate systems are defined at this level, the registration system and the email system.
2. System oriented view – the entire university computer system including email.
3. Actor oriented view – the full information system incorporating both automated and manual processing.

Business Rule: Residency Requirement – In order to complete the residency requirement, the student must be registered as a full time student for one academic year (two consecutive semesters), satisfy the Full Time Student Equivalency Business Rule, or received written approval from the department chair.

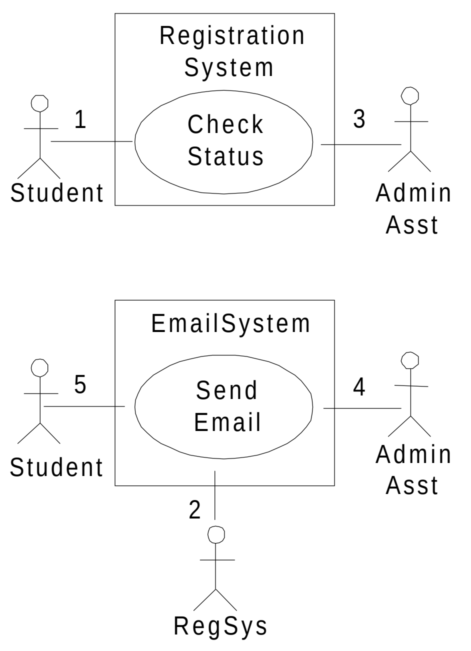
Business Rule: Full Time Student Equivalency – A student may attain the equivalence of full time status for two consecutive semesters by accumulating a sufficient number of class credit hours through regular semester enrollment plus inter-semester courses over a one year period commencing at the beginning of any semester. The inter-semester courses included must be associated with a semesters included during the one year period.

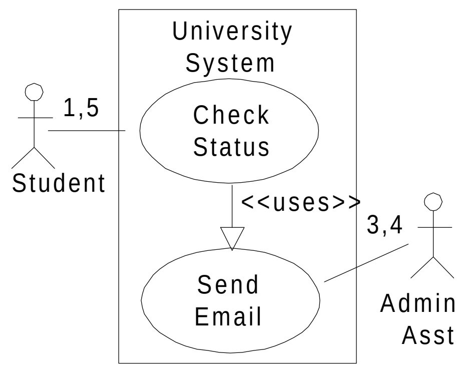
Use Case: Student Checks Residency Requirement Status (CheckStatus)

A student logs onto the university system to check on fulfillment of his residency requirement. The system determines whether or not the student meets the requirements. The system reports the determination of the satisfaction of the requirement to the student. Alternatively, the system responds that fulfillment of residency requirement cannot be automatically determined and that a response will be sent by email later. The system sends an email message to the department administrative assistant to request completion of the workflow. The administrative assistant manually checks the department records and sends the appropriate response to the student by email.

> *Functional Variation*: The system determines whether or not the student meets the requirements, the date of satisfaction, and the method that the requirement was satisfied as stipulated in the residency requirement business rule. The system reports the determination of the satisfaction of the requirement, the date, and method to the student.
>
> *Functional Variation*: If the student has not met the requirement, the system reports the methods of satisfying the requirement and the current status toward each method.

Example 5.1. CheckStatus Use Case and Related Business Rules





(a) (b)

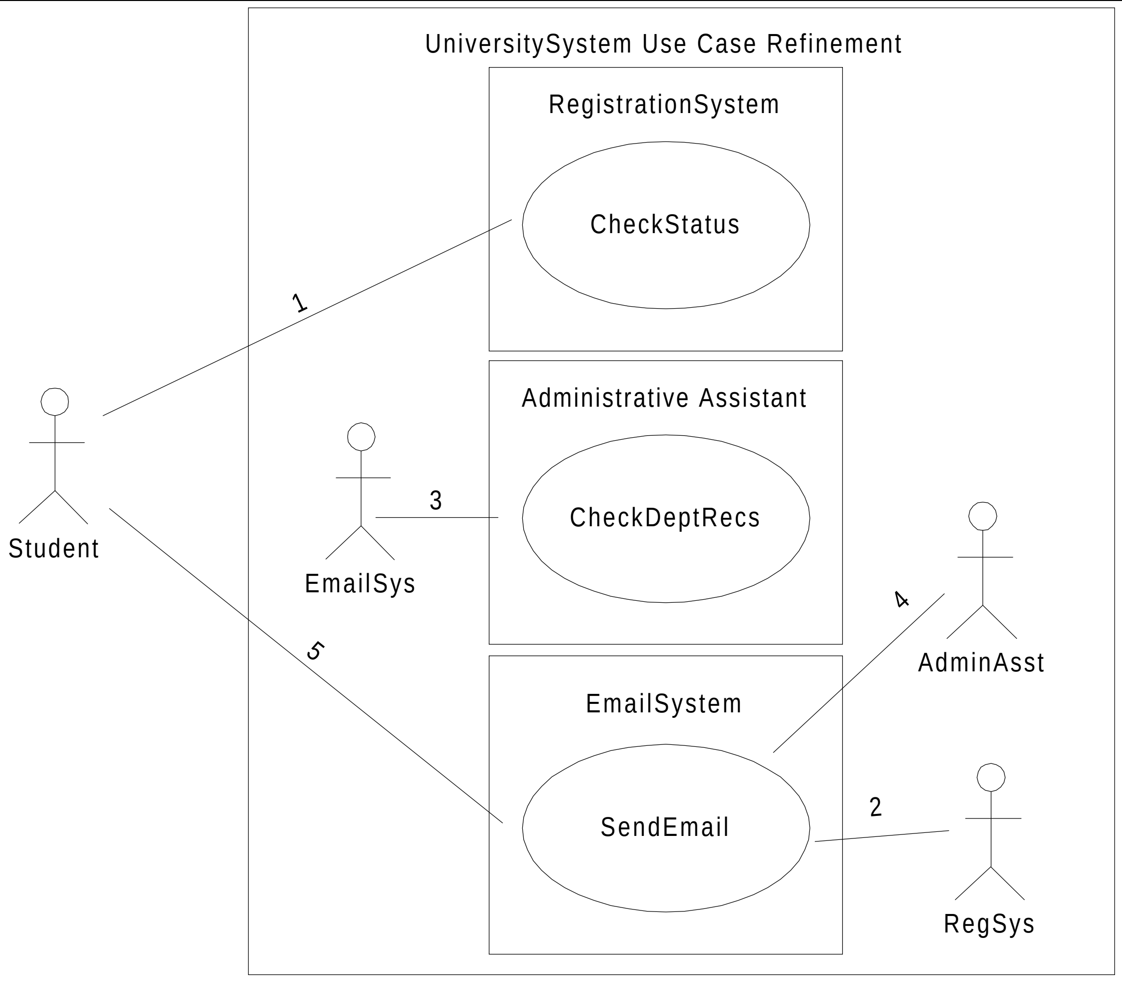
Figure 5.7. CheckStatus – Component View

In the first system boundary case (Figure 5.7a), actual completion of the workflow may occur as the result of interaction between two actors, the *Student* and *AdministrativeAssistant*. Two systems must be modeled to provide an accurate depiction of the workflow, the *RegistrationSystem* and the *EmailSystem*. Consequently, there is no continuity of workflow between the two actors. Although the *Student* has no interest in how the department *AdministrativeAssistant* is notified, these details are modeled as part of the use case. In fact, the actor is the *EmailSystem* and not the department *AdministrativeAssistant*. When the system initiates a message to the *AdministrativeAssistant* via the *EmailSystem*, there is no observable result of value to the *Student*. The inclusion of the *EmailSystem* as an actor primarily serves to highlight the system boundary.

The second system boundary case (Figure 5.7b) includes the email system as part of a larger *UniversitySystem*. It also identifies the department *AdministrativeAssistant* as the secondary actor of the use case. Unlike the first system boundary, the *Student* observes a result of value from the secondary actor in the form of an email response. However, similar to the first system boundary, the inclusion of the secondary actor primarily serves to highlight the system boundary. How the system fulfills its functionality is not the concern of the student. Nevertheless, interaction details are revealed between the system and the department administrative assistant that exposes implementation details. These details do not belong at the same level of abstraction from the perspective of the student actor. If the interactions with the department administrative assistant were separated into its own use case, then the student's use case would include the generation of the email message to the administrative assistant, since a copy of the message will be sent to his email account as well. Nonetheless, the workflow continuity is again lost.

The last system boundary case encapsulates all interactions necessary to yield the complete and observable result for the student as part of an all-encompassing university administration entity. This concept contrasts the notion of a use case model described in the UML Extension of Business Modeling. The UML extension expands the definition of a system to include human workers, but still implicitly allows secondary actors. The system boundary remains the key aspect of use case elaboration. If the system boundary is only perceived to be relevant to the primary actor, then the only distinction that needs to be made is what represents the actor. Everything else being modeled becomes the system. Figure 5.8 illustrates the basic use case and also shows a refined version that reveals use cases for the component entities of the university administration system. The actual use case should only contain steps 1 and 5. The other steps reveal internal structure that should be suppressed at the university administration abstraction level. The numbering of the use case scenario steps can be compared to the same use case actions in the previous case illustrations.

This example shows that the system boundary must be of sufficient scope to capture the workflow process. Case 1 fails to accomplish capturing the entire workflow. Although Case 2 succeeds, it reveals some of the implementation details due to the unnecessary coupling of secondary actors. By employing an actor oriented perspective of the system, secondary actors will only be revealed in lower level refinements of the system, as shown in Case 3. The superordinate/subordinate relationship between use cases is established between the system level use case and use cases for each individual system component, human and machine. The collaboration between model elements of a subsystem package shows how the use case is realized. It is at this subordinate level of use case refinement that secondary actors will be revealed.



Use Case: Student Checks Residency Requirement Status
Scenario: Administrative Assistant makes Determination

1a. A student logs onto the university system to check on fulfillment of his residency requirement.
1b. The system responds that fulfillment of residency requirement cannot be automatically determined and that a response will be sent by email later.
2. The registration system composed an email message for the administrative assistant requesting a determination of the residency fulfillment requirement and instructs the email system send it.
3. The email system delivers the message to the administrative assistant.
4. The administrative assistant makes a determination and composes a reply and sends it by email.
5. The email system delivers the response to the student.

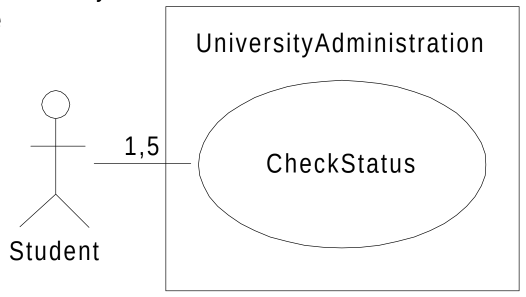


Figure 5.8. CheckStatus – Actor View

5.5. Adaptive Frames

Similar to speech acts, integrating the concept of adaptive frames into use case modeling places additional constraints upon model elements as defined in the UML. However, the incorporation of speech act semantics still represents a structure-oriented perspective, merely shifting the focus of attention from the system to the actor. Adaptive frames provide a behavior-oriented perspective through its focus on scenarios rather than the generalized collective behavior represented in use cases. Whereas the integration of speech acts highlights the system boundary and secondary actor aspects of a use case model, integration of adaptive frames highlights package boundaries and variant scenarios. In order to accommodate adaptive frames into the UML definition of use cases, these key points must be reconciled:

1. Use cases specify one or more sequence of actions
2. Variants (alternative courses of actions) are included as part of a use case, not as a separate use case.
3. A scenario is a specification of a specific sequence of actions. Thus, a scenario describes one particular path through a use case.
4. A use case is realized through collaborations, which in turn define interactions, which in turn are described by scenarios. Thus, scenarios describe realizations of use cases.

A scenario may be defined as a collection of scenarios [Cock97], [KPW97]. However, the UML specification does not recognize a scenario as a first class model element. The closest it comes is the definition of a *UseCaseInstance* as a specifically named sub-type of the *Instance* model element. It represents the actual performance of the use case and implies a realization of the use case through classes and their attached methods that realize the use case's associated operations.

5.5.1. Use Case Attributes

The UML does not describe how its attributes and operations are used to represent the sequence of actions. The only pre-defined attribute is an *extensionPoint*, a list of type *String*. This list merely represents the extension points. They are not the extension points themselves, i.e. the location at which the use case can be extended with additional behavior. Moreover, it does not specify how variants are to be described.

It is also important to make the distinction between *extensionPoint*, which is a use case meta-attribute and the attribute associations that can be attached to a use case. By making the comparison to the Class meta-attribute, this may become clearer. Obtaining the value for *UseCase.extensionPoint* would be similar to determining whether or not a class was abstract by obtaining the value for *Class.isAbstract*. Both describe the model element and are primarily concerned with maintaining well-formed models. An extension point merely indicates that one or more extends relationships exists where the owning use case is the target of the relationship.

Another distinction that should be made along the same lines deals with UML model elements and their instances. The *Class* model element has instances such as Person, Address, and Company. The *UseCase* model element has instance such as PlaceOrder, CheckStatus, and EstablishCredit. Whereas the types of attributes for the Classes will likely be quite different, the attributes for use cases will often be identical for a given project team or domain model. This is why the WAPI-1 specification identifies workflow definition as the root entity. It contains a reference to the format in terms of entity classes and attributes that can be used in the workflow definition. Each format so specified can also be defined as a stereotyped use case within the UML. This is not to say that each use case has an identical structure. The operations for the use case will vary in a manner similar to that of classes. However, the attributes will tend to be same for a given format. For example, the WRM specifies the state or a process instance as being either initiated, running, active, suspended, completed, or terminated. This could easily be translated into an attribute for the all use cases as a *UseCaseStateKind* enumeration.

5.5.2. Attaching Scenarios to Use Cases

In order to comply with the well-formedness rules of the UML, a scenario cannot be a use case. Otherwise its association with the owning use case would be a violation of the constraint that use cases can not have associations to use cases specifying the same entity. Instead, scenarios can be defined as *Class* model elements. They can then be declared a type of attribute of the use case. Providing structured use case representations through attached scenarios offers several advantages:

- Scenarios do not have to be named. Scenarios are instances of the class *Scenario* rather than having a new model element for each scenario as is done for use cases. Use case action sequences and individual use case actions are also instances rather than named model elements.
- Scenarios allow management of finer specification detail. Scenarios keep the use case as a specification model rather than implying realization though associations to actual actions. This

association can come through a dependency relationship as part of a transformation process. Thus, scenarios and its use case actions primarily serve as text management constructs.

- Scenarios facilitate use case evolution. Scenarios enable better management of changes of a use case by easily allowing additional variations to be added or removed. This can even occur dynamically as the system is running if dealing with self-modifying systems.
- Scenarios provide a mechanism for model element to be compatible with multiple standards. They allow definition of use case formats as UML stereotypes that conform to WRM specifications.
- Scenarios provide a mechanism for integrating business rules. Since scenarios are UML C*lass* model elements, they support the notion of templates. Therefore, parameters can be integrated into the structures. This provides the support for business rule binding. Moreover, support is also provided for the different data types that result from use case variations that accommodate actor role subtypes.

Scenarios can transcend varying levels of abstraction. In order to maintain consistency with the integration of speech acts, it is appropriate that alternative sequences of actions not perceived from the outside should be suppressed. If all scenarios adhere to this guideline, then each becomes the description of a use case variant. Adaptive frames place one additional constraint upon a use case by requiring that the basic course of actions, referred to as the archetype scenario, does not contain any conditional branching. Once an archetype scenario has been declared, all alternative courses of action, exceptional behavior, and error handling can be completely defined in terms of scenarios built by describing their differences from the archetype scenario. Adaptive frames run contrary to strict inheritance, where structural and behavioral features can be replaced or added, but not deleted. Instead of trying to coerce a contrived sequence of actions upon each alternative scenario, only the actions explicitly declared as common actions between scenarios are reused. This reuse may also occur across use cases, since action sequences of one or more actions can be contained in their own frame for reuse. Once they have been defined, these frames may then be grouped into packages. Using fine grained frames consisting of actions and action sequences provides greater opportunities for use case reuse than larger grained generalization stereotypes defined in the UML.

5.5.3. The «adapts» Stereotype Relationship Between Scenarios

The relationship between the archetype scenario and the variant scenario is established through an «adapts» UML stereotype. Part of the motivation for this stereotype is to replace occurrences of «extends» and «uses» that might occur if this stereotype did not exist. Therefore, the introduction of the «adapts» stereotype between scenarios must be reconciled with the pre-defined UML use case generalization stereotypes. The following observations are made based on the UML definitions:

1. The ordering of the sequence of actions can not be altered for an «extends» or «uses». The UML definition is silent on whether or not all actions must be included or how any restrictions are applied to the basic sequence of actions or variants.
2. The «extends» and «uses» relationship allows a name to be attached to a use case.
3. The key distinction between «extends» and «uses» is that «uses» is an unconditional inclusion and «extends» is conditional. Associating «uses» with common behavior and «extends» with exceptional behavior appears to be a holdover from past semantics that have no such semantic constraints in the current UML specification.
4. From the perspective of the use case that gets additional behavior added to it, the direction of generalization is opposite for «extends» and «uses».

An «adapts» stereotyped relationship between a variant scenario as the source and an archetype scenario as the target results in a set of adaptive frames. Each variant can also be adapted. Without the «adapts» stereotype or similar construct as part of the UML meta-model, the modeler must choose between no reuse or contrived reuse through «extends» and «uses», resulting in model elements that may be more appropriately defined as "abstract use case fragments." In the UML Extension for Business Modeling, a use case is prohibited from being partitioned over several use case packages. The integration of adaptive frames conforms to this restriction by directly associating a use case with its archetype scenario and then incorporating all variant scenarios as an adaptation of this baseline use case. Thus all parts of a concrete use case is contained in a single package, regardless of the fact that it may adapt through a web of frames contained in other packages.

5.6. Related UML Use Case Model Constructs

In order to complete our conceptual foundation for definition of the adaptive use case meta-model, it is necessary to digress momentarily to introduce related UML concepts that will be used in the model. These are the extension mechanisms, predefined standard elements that are not considered first class model elements, and semantics for the *Actor, Template,* and *Binding* model elements with respect to how they relate to use cases.

The UML provides three built-in extension mechanisms, the *Stereotype*, *Constraint*, and *TaggedValue*, that enable new kinds of modeling elements to have distinct semantics, characteristics and notation relative to the built in UML modeling elements specified by the UML metamodel. *Stereotypes* facilitate the addition of “virtual” UML meta-classes with new attributes and constraints. A new graphical representation may also be introduces to distinguish stereotyped model elements. Although a stereotype shares the attributes, associations, and operations of its base class, it may have additional well-formedness constraints as well as a different meaning and attached values. A *Constraint* or *TaggedValue* associated with a particular stereotype is used to extend the semantics of model elements classified by that stereotype. A constraint is a semantic condition or restriction represented as a Boolean expression. Tags serve as “pseudo-attributes” of the stereotype to supplement the real attributes supplied by the base element class. The values permitted to such “pseudo-attributes” can also be constrained.

By defining a small set of additional subtypes to the basic use case concepts, the well-formedness of speech acts and adaptive frames can be defined formally, and subsequently mapped to the dynamic semantics of use cases. In addition, the use case extensions eliminate ambiguities that might otherwise arise in the interchange of use case models between tools.

5.6.1. UML Use Case Standard Elements

The UML defines three stereotypes as part of the use case package. The two generalization stereotypes have already been discussed. The third stereotype is «useCaseModel». It describes the function requirements of a system is terms of the use cases and interactions with actors. Since only actors, use cases and their associations are permitted, the use case model provides only a high level functional view of a system. No additional tagged values or constraint are defined.

The UML specification for a use case defines only one attribute, the list of extension points. There are two attributes associated with the «extends» association, a reference to an extension point and a condition. Although the semantics permits extending the behavior at more than one extension point and including more than one condition, there is no clear specification for this. Only the presence of a use case extension point is represented in the UML use case model.

5.6.2. Actor

An *Actor* defines a set of roles played by a user, one role for each use case with which it communicates. These roles are subsumed by the Actor model element at the specification level because UML semantics prohibit an Actor from containing other *Classifiers*. However, at the realization level, these roles become manifest through participation in collaborations as *ClassifierRoles*.

In the simplest case, a single Actor interacts with all use cases. When more than one actor is being modeled to interact with the same use case, then both Actors will be realized through the same collaboration, provided that both utilize the same operation. However, it is quite possible that the data flow between different Actors and a single use case can occur resulting in different signatures for the operations. If different Actors communicate with the same set of use case, varying only by the data exchanged, then differentiation of these Actors can be accomplished through an inheritance hierarchy for an Actor, as illustrated in Figure 5.3. However, this leads to a potential combinatorial explosion of Actor sub-type permutations if many facets are involved in establishing the data structures associated with a use case operation. Alternatively, adaptive frames may also be implemented for defining Actor subtypes. This allows the modeler to focus on the differences in data exchanges between the Actor subtypes and the entity being specified by the use cases. In this manner, the adaptive Actor becomes associated with a parameter for a use case template. Also, the data structure can be generalized in the use case interface and refined through sets of scenarios attached to the use case through the adaptive frame mechanism.

5.6.3. UseCase Template

A UseCase template is a UseCase that contains one or more template parameters. In our approach, all use case templates consist of a single parameter of type *Set(BusinessRule)*. A template parameter denotes which parts of the use case are parameterized. Each business rule contained in this set gets attached to a defined slot in the use case. When a set of Actors becomes associated with the use case, the use case becomes complete by replacing the parameterized placeholders with the supplied arguments belonging to each actor subtype. At this point it becomes fully subject to well-formedness rules. This is accomplished by effectively duplicating the UseCase template and adding it to the use case model with the parameter substitutions as if it had been modeled directly. This is similar to what occurs with adaptive frames.

This approach shares many similarities with the parameterization and use case template technique proposed by Jacobson, *et al.* [JGJ97]. Both approaches require that when abstract use cases are reused, they have to be specialized by selecting among alternatives that can be attached at defined variation points. These use case variation points are identified through special notation. Furthermore, both approaches conform to the template and binding semantics of the UML specification. The binding process generates a completely new use case.

However, there are several significant differences. First, the Jacobson approach makes a distinction between parameterization and use case templates. Parameters are mostly used to define relatively simple and well-defined variability by replacing the parameter slots with some text or references. Although their approach identifies several variability aspects, no distinction is made within the actual semantics of the parameterization mechanism. The approach presented in this thesis elaborates and extends the simple parameterization concept. We first create a dichotomy between the two major entities of a use case dialog, namely the actor and the system. The actor influences, shown in Table 5.2, represent decisions made by the actor which primarily determine the actual scenario that is performed. The system influences shown in Table 5.3 are decisions that are made by the system to perform its responsibilities in the use case dialog. They form the basis for the scope of scenarios that must be defined for the use case.

Table 5.2. Actor Influences on Use Case Variation

Variation Aspect	Mechanism	Renew Library Item Example
User	Actor sub types	Child, Adult, Employee, Another Library
Interface	Semantic interface; use case dialog map	Phone, Web-based
Referenced Entity	Parameter	Book, Magazine, Video
Alternative action	Scenario condition	By Catalog #, by Title
Optional action	Scenario condition	Confirmation

Table 5.3. System Influences on Use Case Variation

Variation Aspect	Mechanism	Renew Library Item Example
Exception action	Scenario; condition	Invalid entry, response timeout (performance), system user overload (scalability)
Business Rule Derivation	Inference, Computation	Overdue
Business Rule Action	Authorization, Condition, Integrity Constraint	On-hold for another

The second difference is where the parameters are generated. This approach attaches parameters to scenarios rather than the entire use case. This provides a much finer grained control over each of the variation aspects by allowing direct mapping to a subset of the behavior of the use case that is relevant to the variability. It also allows for coupling the variability aspects for the actor and the system. This is especially important for exception actions that deal with error-detection and recovery. For example, one use case for a library system would be to renew a library item. Exception handling will be affected by the type of user (adult patron or employee), the interface (phone or web), the type of item (book or video), any alternative choice by the user (enter title or catalog number), and any optional actions requested (confirmation). Each of these aspects of the user's profile and decisions affects the appropriate response by the system when an exception occurs, such as making an invalid entry. Other possible exception actions, such as a response timeout or being held in queue due to exceeding simultaneous user capacity, are affected by the user profile only. Nonetheless, each exception action must be coupled with the corresponding actor influenced variability aspects.

The third difference addresses how interdependent parameters are dealt with. The previous use case example, *Renewing A Library Item*, introduced a level of complexity that is treated in a separate variability mechanism in the Jacobson approach. There, a generator tool is used to process the use case template. The replacement text attached to each parameter slot is treated as executable procedures or scripts. Processing the code generates a completely new concrete use case. It may also produce a set of analysis types. The adaptive use case approach allows the individual scenarios to be realized in concrete form rather than the entire use case. Through the definition of a set of business rules as the parameter, each permutation of actor variability can be mapped to appropriate system responses and behavior. Furthermore, parameters can be applied incrementally. Only details under current consideration need to be investigated.

This permits more control over exploring the ramifications of introducing additional variation points into the use case by isolating the set of parameters that are directly affected by the new variant.

The fourth difference is concerned with how use cases are specialized. The Jacobson approach relies on the «uses» and «extends» generalization stereotypes. The «extends» mechanism adds actions to a use case that is already complete. The «uses» mechanism shares common behavior definitions. Although this provides conformance with the UML specification, it carries the limitations previously discussed. Our approach emphasizes the dialog aspect of the use case. A generalized actor may communicate with the system through a generalized use case. Through restriction of the scenarios in which an actor subtype can participate, true specialization of use cases results without the need to define new use cases for each actor subtype. This is done through clustering of specialized scenarios that are only concerned with a single actor subtype. Thus, composition is used to associate specialized use case behavior with a particular actor subtype. Scenario composition conforms to the UML well-formedness rules while providing a mechanism to refine the relationship between each actor subtype and the use case. This represents refinement of the collaboration similar to the Catalysis method, which has been integrated into the UML through its collaboration refinement mechanism [DW97]. Thus, the adaptive use case model extension is strengthened by its affinity with existing UML semantics.

5.6.4. UseCase Binding

A *UseCase Binding* contains the list of arguments that replace the *Set(BusinessRule)* parameters of the UseCase template to create the fully specified UseCase. Each business rule is composed of a structural assertions and derivations that identify the subtype of the Actor, the data typed exchanged between the Actor subtype and the entity within the context of the use case being created from the template. This strategy promotes a covariant type policy. *Covariance* can accept more specific information than the superclass at the expense of substitutability. A covariant type policy allows the method argument type for a subclass to be a subtype of the method argument type for its superclass. This type policy is in contrast to *contravariance,* which assures substitutability of instances with subclasses. A contravariant type policy requires that the method argument type for a class and subclass are the same (conservative contravariance) or that the method argument type of the subclass be a superclass of the method argument type (regular contravariance).

Although a covariant type policy seems to expand the degrees of freedom, it relies on additional knowledge to assure a safe downcasting, or matching of compatible types. In order to have a well-formed use case, a meta-object query can provide one part of the solution. In other words, the scenario must know which type of actor it is interacting with. The property of homomorphism, or the parallel mapping of two classes, is also part of the practical solution to obviate this restriction. This allows for a set of scenarios to be exclusively associated with a particular actor sub-type.

The actor type is used in the *VariantScenarioActionSplice.condition* attribute as a parameter. This is represented as a BooleanExpression that evaluates to a Boolean. It can be represented in the Object Constraint Language (OCL) which is defined in the UML specification. For example, using the renew library item example described earlier in this section, the bindings for the actor and the data structure of the referenced entity would be declared as follows:

```
oclType(Actor) = oclIsTypeOf(<BusinessRule.actorType>)
operation: renewLibraryItem(<BusinessRule.itemTypeData>)
```

The actual generation of the OCL constraints would be automatically generated from business rule templates. These templates would specify the actor subtypes, referenced entities, and other conditional expressions that are necessary to create the desired scenarios to fully describe the use case.

5.7. The Adaptive Use Case Meta-Model

This section provides the definition of each new stereotype, taggedValue and constraint needed to model the Adaptive Use Case mechanism. It represents the synthesis of each of the major components introduced earlier in this chapter. The stereotypes described in this section overcome the ambiguities with respect to extension points and variability mechanisms in the UML specification. The presentation in this section follows the same structure as the Business Modeling Extension and Objectory Process Extension to the UML.

5.7.1. Introduction

This document defines the UML Extension for Adaptive Use Case Modeling for Managing the Evolution of Business Domain Architectures, defined in terms of the UML's extensions mechanisms, namely, Stereotypes, TaggedValues, and Constraints. See the UML Semantics document for a full description of the UML extension mechanisms [UML97].

This section describes stereotypes that can be used to tailor the use of UML for managing the evolution of a business domain architecture. All of the UML concepts can be used for domain engineering, but providing adaptive use case stereotypes introduces a common terminology and approach for common situations that occur during the evolution of a business domain architecture. Note that UML can be used to model different kinds of systems, including both software systems and real-world organizations. This extension is intended to bridge the semantic differences between these two types of systems. Systems boundaries are defined in terms of actor-oriented views for a given level of abstraction.

This document is not meant to be a complete definition of the Adaptive Use Case modeling technique, but it serves the purpose of registering this extension, including its icons.

5.7.2. Summary of Extension

The adaptive use case extension consists of twenty-six stereotypes and two tagged values, as shown in tables Table 5.4 and Table 5.5. This extension does not currently introduce any new Constraints, other than those associated with the well-formedness semantics of the stereotypes introduced.

Model Management Stereotypes. A business domain architecture is represented through several different, but related models. These models are characterized by the view focus of the stakeholder that they represent. The stakeholder view establishes the context for defining the system boundary. The system boundary is progressively narrowed for each of these stakeholders: user, customer, architect, designer, and implementer. All models may be considered analysis models from the perspective of the stakeholder when looking outward, since the models only reveal collaboration details external to the system definition at the current refinement level for the stakeholder. However, when looking inward to a narrower stakeholder view, internal implementation details are revealed such that these models may be consider design models from the perspective of the higher level stakeholder.

Class Stereotypes. Each model element described is either a stereotyped *UseCase* or an attribute either directly or indirectly contained in the use case. Each attribute described represents a tagged value.

Notation. All stereotypes contained in this extension are indicated with stereotype keywords in guillemets («stereotypeName»). Alternatively, several class stereotypes may be shown with special icons.

Although no other UML extensions are required, the following predefined UML stereotypes are used: *derived, facade, framework, import, stereotype, stub, system, topLevelPackage*, and *useCaseModel*.

Table 5.4. Stereotypes

Meta-Model Class	Stereotype Name
Model	AdaptiveUseCaseModel
Package	AdaptiveUseCaseSystem
Package	AdaptiveUseCasePackage
Package	ConfigurationItem
Package	Configuration
Model	UserUseCaseModel
Model	CustomerUseCaseModel
Model	ArchitectUseCaseModel
Model	DesignerUseCaseModel
Model	ImplementerUseCaseModel
Use Case	NarrativeUseCase
Class	NarrativeScenario
Use Case	AdaptiveUseCase
Class	Scenario
Class	ArchetypeScenario
Class	VariantScenario
Class	VariantScenarioAction
Class	BusinessRule
Enumeration	RelativePlacementKind
Class	UseCaseAction
Class	UseCaseActionSequence
Model Element	UseCaseTemplate
Association	Adapts
Class	WorkflowParticipant
Class	WorkflowProcess
Class	WorkflowActivity

Table 5.5. Tagged Values

Tagged Value Name	Applies to Stereotype
mapping	NarrativeUseCase NarrativeScenario
narrativeDesciption	NarrativeUseCase NarrativeScenario

5.7.3. Adaptive Use Case Model Stereotypes

An *AdaptiveUseCaseModel* is a stereotyped «useCaseModel» in which the top-level package is an *AdaptiveUseCaseSystem*. It describes the workflow *WorkflowProcesses* in a business domain in terms of *WorkflowParticipants*, *Activities* that are performed by the *WorkflowParticipants* and subsystems that are invoked by workflow participants or other subsystems.

An *AdaptiveUseCaseSystem* is a stereotyped topLevelPackage for an *AdaptiveUseCasePackage*.

An *AdaptiveUseCasePackage* is a package containing either directly or indirectly containing at least one *AdaptiveUseCase*, plus *Actors* and the relationship between the *Actors* and the *AdaptiveUseCase*. Although an adaptive use case model element belongs to a single package, it constituent parts may be partitioned over several packages. Therefore, an adaptive use case package may also contain *ConfigurationItem* packages, «stub» packages, «facade» packages, or «framework» packages consisting of *Actors*, relationships between *Actors* and *AdaptiveUseCases*, or any other stereotyped model element defined in this extension.

A *ConfigurationItem* is a package that that contains *AdaptiveUseCase* constituent parts such as a *Scenario*, *UseCaseSegment*, or their respective subtypes. A *ConfigurationItem* package is primarily intended to provide version control.

A *Configuration* is a package that that contains one or more *ConfigurationItem* packages.

5.7.4. Stakeholder Model Stereotypes

A *UserUseCaseModel* is a model that contains exactly one *Actor*, one or more *UseCases* that specify the complete functionality of a system from the viewpoint of a single actor, and the relationships between the *Actor* and the *UseCases*. These model elements may be contained either directly by the model or indirectly through another contained package.

A *CustomerUseCaseModel* is a model that contains all *Actors* and *UseCases* for a system, plus the relationships between the actors and the use cases. These model elements may be contained either directly by the model or indirectly through another contained package. This model represents the UML context for a top-most level use case model and is stereotyped here only for completeness. No additional semantics are attached to this stereotype.

An *ArchitectUseCaseModel* is a model that contains exactly one customer use case package, plus the use case refinements for the system in the form of use cases for all components and *WorkflowParticipants* as well as the collaborations among them to fulfill the superordinate use case. This model represents the viewpoint of the top-most level use case model in the UML specification plus the specification part of all subsystems contained in the realization of the UML top-most level use case model.

A *DesignerUseCaseModel* is a model that contains all actors and use cases for a subsystem, the relationships between the actors and the use cases, plus all superordinate use cases and actors up to the highest level. The designer use case model must not be a bottom-most level package. Specifically, at least one *ImplementerUseCaseModel* must exist that specifies the collaborations representing the realization of the subsystem being modeled.

An *ImplementerUseCaseModel* is a model of a bottom-most level package that specifies all actors and use cases for a subsystem or class, the relationships between the actors and the use cases, plus all superordinate use cases and actors up to the highest level. No packages or models may exist that specify the subsystem or class being modeled.

5.7.5. Adaptive Use Case Stereotypes

A *NarrativeUseCase* is a stereotyped use case that contains a narrative description of a sequence of actions. It represents a high-level use case that may contain conditional statements, such as if-then-else statements. It may contain an optional list of *NarrativeScenarios.* If so, the narrative text is a summary abstraction of the component *NarrativeScenarios*. The tagged value, *mapping* of type *Expression* contains procedure for performing this summary. The *NarrativeUseCase* is an appropriate representation for communicating requirements and functionality among domain experts, users, and developers. The entire narrative description is stored in a single attribute *narrativeDesciption* of type *String*.

A *NarrativeScenario* is a stereotyped scenario that contains a narrative description of a sequence of actions derived from *Scenario.* The tagged value, *mapping*, of type *Expression* contains procedure for performing this summary. *NarrativeScenario* may not contain any condition statements. It is generally used to fully describe an instance of *Scenario.* The entire narrative description is stored in a single attribute *narrativeDesciption* of type *String.*

An *AdaptiveUseCase* is a stereotyped use case that contains an ordered set of actions in the form of a basic course of action as an *archetypeScenario* (of type *Scenario*), plus optional variant courses of actions as variantScenarios (of type list of *Scenario*). An adaptive use case represents a single workflow dialog. It is a specialization of *UseCase*, since only one (the initiating) actor is permitted to be associated with the use case. All interactions with other actors are suppressed, although they may become known at a more refined level of the use case. The interface for an *AdaptiveUseCase* is the set of all interfaces for the use case with a single actor.

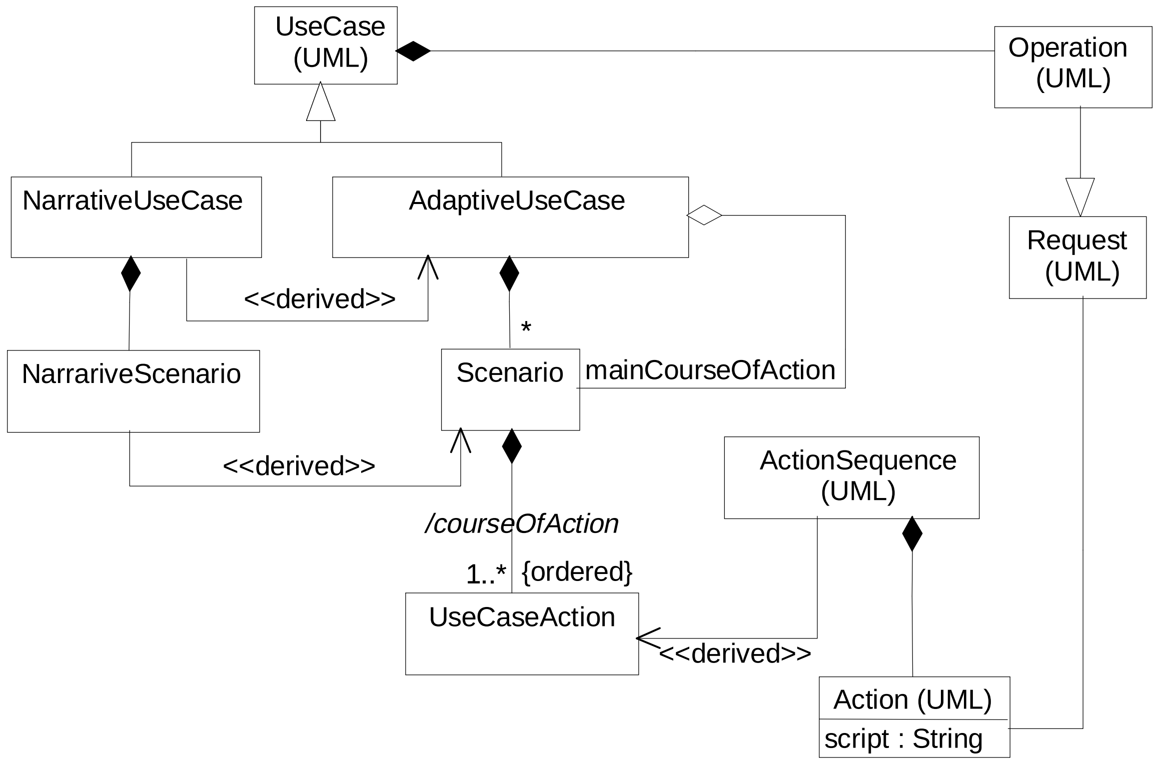


Figure 5.9. Adaptive Use Case Stereotypes

5.7.6. Workflow Stereotypes

A *WorkflowParticipant* is a stereotyped *ClassifierRole* participating in a *Collaboration* that realizes a superordinate use case for a subsystem. A *WorkflowParticipant* may also play the role of *Actor* when invoking a use case for the system at the same level of abstraction.

A *WorkflowProcess* is a stereotyped *AdaptiveUseCase* that is extended to include *Activities* performed by *WorkflowParticipants*. For any given viewpoint and level of abstraction, a *WorkflowProcess* includes all *UseCaseActions* of all superordinate use cases. It should be noted that a *WorkflowProcess* is similar to the predefined «process» standard element of the UML. A *WorkflowProcess* may be realized by a «process» active class, but other realizations are also possible.

A *WorkflowActivity* is a stereotyped *UseCaseAction* that is performed by a *WorkflowParticipant*.

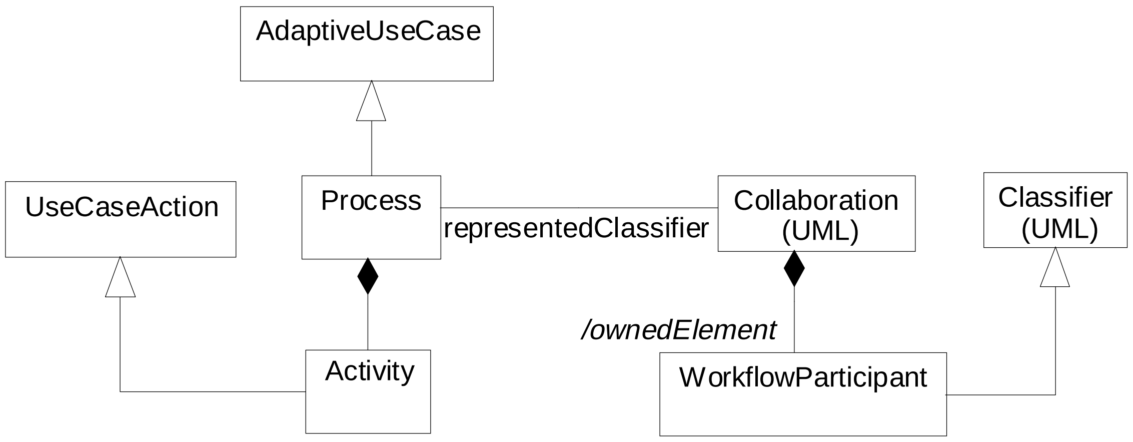


Figure 5.10. Workflow Stereotypes

5.7.7. Scenario Stereotypes

A *Scenario* is a stereotyped class representing a typed attribute of a use case. It represents an ordered set of actions contained in its derived attribute *courseOfAction* (of type list of *UseCaseAction*). *Scenario* may not contain conditional branching, however iteration is allowed if it is contained in a single action. Refinements to *Scenario* may reveal conditional branching only as parts of subordinate use cases. The commencement and completion of each action is represented as a (potential) extension point. Each Scenario has an associated interface.

An *ArchetypeScenario* is a specialization of *Scenario.* It contains an ordered sequence of actions through its list of use case segements. *ArchetypeScenario* is typically used to represent the basic course of actions for a use case. This list of *UseCaseSegments* is used to derive the attribute *courseOfAction* declared in *Scenario.*

A *VariantScenario* is a specialization of *Scenario* representing an ordered sequence of actions that represents an alternate course of actions. The *VariantScenario* is defined in terms of another scenario (referred to here as the base scenario) belonging to the same use case – either an *ArchetypeScenario* or another *VariantScenario*. The *VariantScenario* adapts the base scenario through a set of *VariantScenarioActions*. Each *VariantScenarioAction* consists of a condition, a starting extension point, and an ending extension point plus a set of zero or more actions. If the condition evaluates true, the actions associated with the *VariantScenarioActions* are spliced between the two extension points.

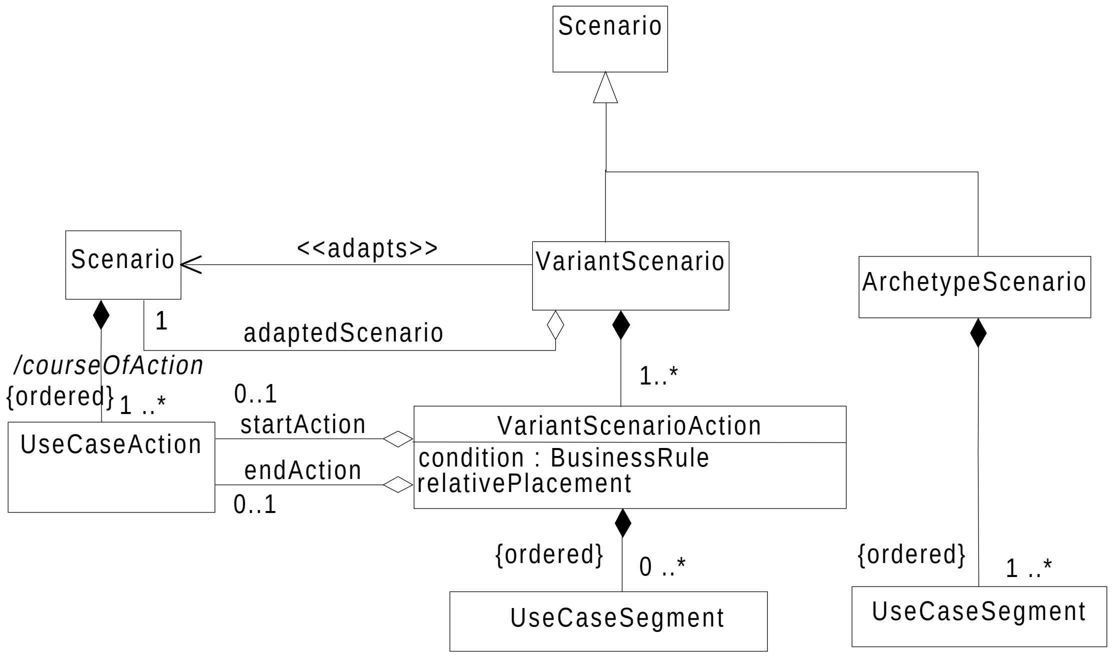


Figure 5.11. Scenario Stereotypes

An *VariantScenarioAction* is a stereotyped class representing an ordered list of use case actions contained in a list of *UseCaseSegments* that replace actions in the *Scenario* identified as the base scenario by *VariantScenario* that owns it. The pre-condition for following this variant course of action is contained in the attribute *condition* of type *BusinessRule*. The precise placement of the starting and ending extension point references are specified in the *relativePlacement* attribute of type *RelativePlacementKind*. In general, if both extension points are the same, then this represents extended behavior. If the extension points are different, this represents replaced behavior. If the set of actions is empty, this represents removed behavior.

A *RelativePlacementKind* is a stereotyped Enumeration that defines how the actions associated with a *VariantScenarioAction* will be inserted into the base scenario identified by a *VariantScenario.* The complete set of values for *RelativePlacementKind* is:

replace	The actions identified are removed, as well as any intervening ones and replaced by the new actions, if any. This translates to before starting action and after ending action. This is the most common usage.
remove	The actions identified are removed, as well as any intervening ones.
inject	New actions are attached before the starting action. The ending action is ignored. This is identical to a *replace* where the ending action immediately precedes the starting action, except that the condition will not be reevaluated. However, if the condition also included this additional guard, then it could be modeled with a replace. *Inject* is preferred, since the intention is clearer.
prepend	New actions are attached to the beginning before any actions in the base scenario. Starting and ending actions are ignored. This is a special case of *inject.*
append	New actions are attached to the end after all actions in the base scenario. Starting and ending actions are ignored.
regress	New actions are attached before the starting action. The action sequencing resumes prior to the starting action. In other words, the ending action must precede the starting action.

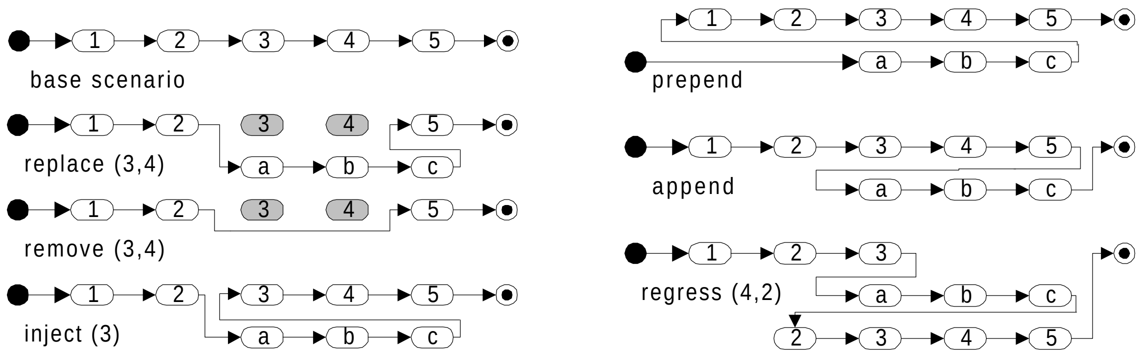


Figure 5.12. RelativePlacementKind Examples

5.7.8. Use Case Segment Stereotypes

A *UseCaseSegment* is a stereotyped class representing an abstract generalization of *UseCaseAction* and *UseCaseActionSequence.* This model element defines the interface for both specialized elements. This permits packaging of reusable use case actions into sequences conforming to the Composite design pattern.

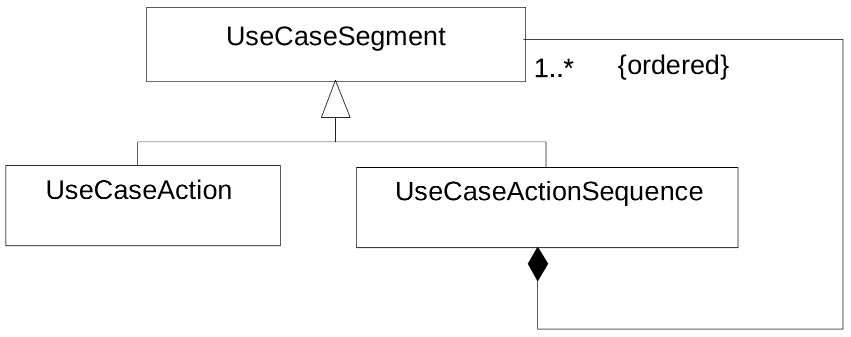


Figure 5.13: UseCaseSegment Stereotypes

A *UseCaseAction* is an atomic action expressed as a narrative statement of type *String*. Either the action is performed in its entirety or it is not performed at all. A *UseCaseAction* may be mapped to the UML model element ActionSequence. *UseCaseAction* is a specialization of the *UseCaseSegment* model element.

A *UseCaseActionSequence* is an ordered list of *UseCaseSegment*. This model element is typically use to package reusable use case actions into any level of functionality from fine grained to large grained.

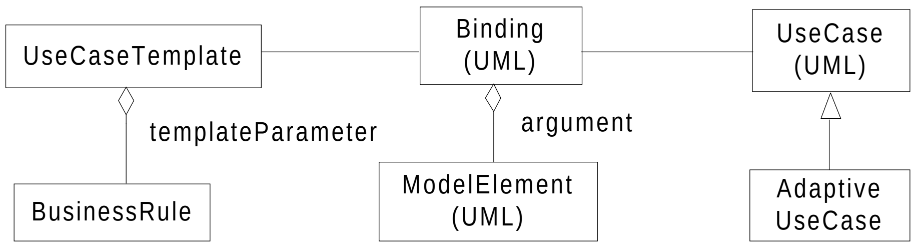


Figure 5.14. UseCaseTemplate Stereotypes

5.7.9. Use Case Template Stereotypes

A *UseCaseTemplate* is a *UseCase* with one or more unbound formal parameters. The parameters may be located in any of the extension model elements described in this section. Actions and extension point conditions can also be defined in terms of the formal parameters, so that they too become bound when the template itself is bound to actual values.

A *BusinessRule* is a stereotyped class representing an ordered set of derivations, structural assertions, and behavioral assertions. Within the context of a *VariantScenarioAction*, the *BusinessRule* typically refers to a *BooleanExpression*. However, a business rule can represent of complex web of derivations, structural assertions, and behavioral assertions. For example, priority of evaluating conditions for extension points may be specified, although these would not be apparent within the context of any single *VariantScenarioAction* condition. Business rule templates can be utilized to generate this rule set of interconnecting rules.

A *BusinessRuleTemplate* is a *BusinessRule* with one or more unbound formal parameters. The parameters must be located the *BusinessRule* model element. An unbound business rule parameter will evaluate to one or more derivations, structural assertions, and behavioral assertions.

5.7.10. Association Stereotypes

An *«adapts»* relationship is a relationship between two model elements that is based on adaptive frame technology. It is a stereotyped dependency relationship that permits reuse of the target model element without the restrictions of strict inheritance imposed on the source element in a generalization relationship. An *«adapts»* relationship allows for features in the source element to be added, replaced, or deleted to the target element feature set when describing the source element.

5.7.11. Tagged Values

Four tagged values are attached to stereotypes that have been defined in this extension. It is expected that a model analysis tool will utilize these tag-value pairs to support the full range of features of these model element extensions.

A *mapping* applies to both a *NarrativeUseCase* and a *NarrativeScenario*. For its usage in *NarrativeScenario*, it specifies the procedural logic needed to translate a *Scenario* model element into a narrative text representation. The mapping for this derivation is rather straightforward. All parameter bindings are resolved and all nested use case actions are flattened to create a narrative sequence of action statements. *NarrativeScenario* may not contain any condition statements. For its usage in *NarrativeUseCase*, it specifies the procedural logic for summarizing the narrative text representations for each *NarrativeScenario*. In its simplest form, it is a lossless concatenation of the narrative text representation for each contained *NarrativeScenario*. The entire narrative description is stored as type *String*, but represents type *Expression*.

A *narrativeDescription* also applies to both a *NarrativeUseCase* and a *NarrativeScenario*. It captures the narrative text representation resulting from performance of the transformation specified in the *mapping* tagged value for the respective model element. The entire narrative description is stored as type *String*.

5.7.12. Workflow Process Diagram

The workflow process diagram provides for a graphical representation of use case models from the viewpoint of the architect, designer, and implementer. Specifically, a *WorkflowProcessDiagram* is a stereotyped *ViewElement* that presents the set of *ModelElements* contained in an *ArchitectUseCaseModel*, *DesignerUseCaseModel*, or *ImplementerUseCaseModel*.

Semantics. A workflow process shows the collaboration among use cases that refine superordinate use cases through an arbitrary number of layers of abstraction. It is intended to demonstrate prototypical behavior that establishes a context for the bottom-most subsystems of the model. It is also possible to show behavioral exception flows, but the intent of this type of diagram is to provide a concise representation of context for a single scenario in order to minimize the number of graphic elements in the diagram. Therefore several diagrams may be generated from the same set of cascading use case refinements.

Workflow participants, which are stereotyped actors, are used within the top-most system abstraction level. Actors are only presented outside the top-most system boundary. A workflow participant may play either the actor or system role of a use case dialog that represents a use case for a subsystem. The same workflow participant may play both roles, provided that they occur in different use case dialogs.

Several different versions of the workflow process diagram may be utilized. The standard format merely identifies the subordinate use cases that comprise each superordinate use case up to the top-most level. No collaborations are shown. Other versions highlight the process flow and design patterns.

If the process flow is being presented, then four states, each conforming to a speech act, may also be represented. The first state is the result of a directive being made by the actor, such as a request to provide some service. The second state results from a commissive, which represents the system committing to provide a service. The third state results from a declaration by the system that the service has been completed. The fourth state is an assertive by the actor that the service has been provided satisfactorily. Each state is preceded by an activity phase: preparation, negotiation, performance, and acceptance. Each phase may be refined through one or more use case dialogs.

Design patterns may also be represented in a workflow process diagram. Such representation is more precise in a workflow process diagram than in the UML collaboration diagram because a workflow participant is partitioned over several use case dialogs, corresponding to each role played by the participant. Thus, only those aspects of a classifier (i.e. the classifier's role) are captured. Furthermore, since several levels of abstraction are possible, a recurring theme of design patterns may emerge. A developer using the diagram can carry this meta-pattern of pattern usage through to realizations of subsystems that are represented as black-box subsystems at the bottom-level or abstraction within the model.

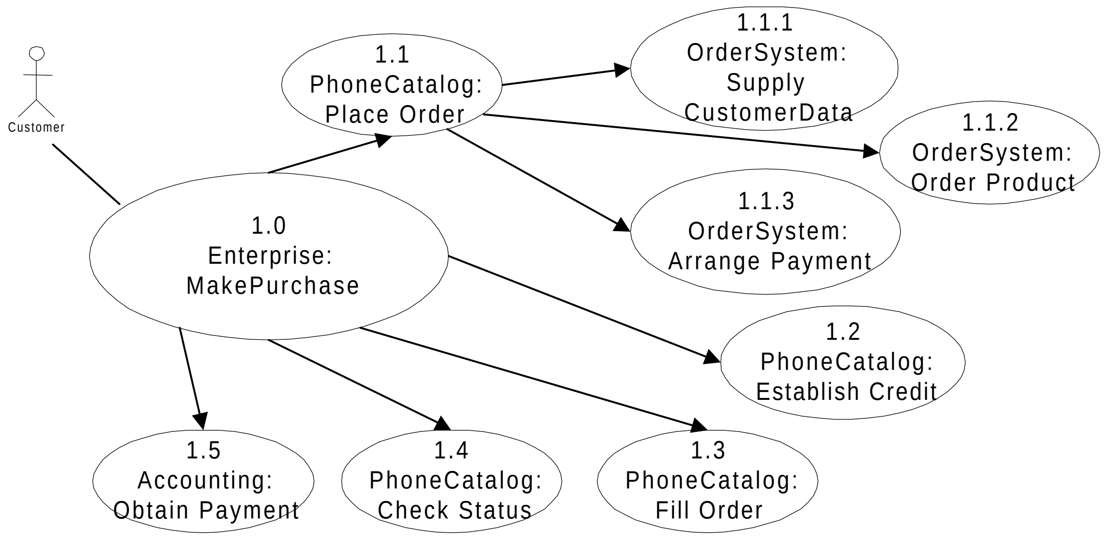

Customer
1.0 Enterprise: MakePurchase
1.1 PhoneCatalog: Place Order
1.1.1 OrderSystem: Supply CustomerData
1.1.2 OrderSystem: Order Product
1.1.3 OrderSystem: Arrange Payment
1.2 PhoneCatalog: Establish Credit
1.3 PhoneCatalog: Fill Order
1.4 PhoneCatalog: Check Status
1.5 Accounting: Obtain Payment

Figure 5.15. Workflow Process Diagram

<u>Notation.</u> A use case dialog is shown as a standard ellipse notation for a use case as defined in the UML. However, the use case name should be preceded by the package name and separated by a colon. Actors that appear outside the top-most system abstraction also use the standard stick-man figure. Solid

arrow lines connect the use cases dialog ellipses. Dashed arrow lines connect the use cases dialog ellipses that appear at different levels of abstraction. Each use case is assigned a number based on its refined level of abstraction from to top-most level and the sequence number of the use case action that invokes it. A legal numbering format convention is used, such as the section numbering scheme used in this document.

Presentation Options. Textual annotation can be included. The annotations would present the text of each use case action involved in the scenario being represented in the diagram. It should following the same legal numbering convention used to annotate the use case ellipses.

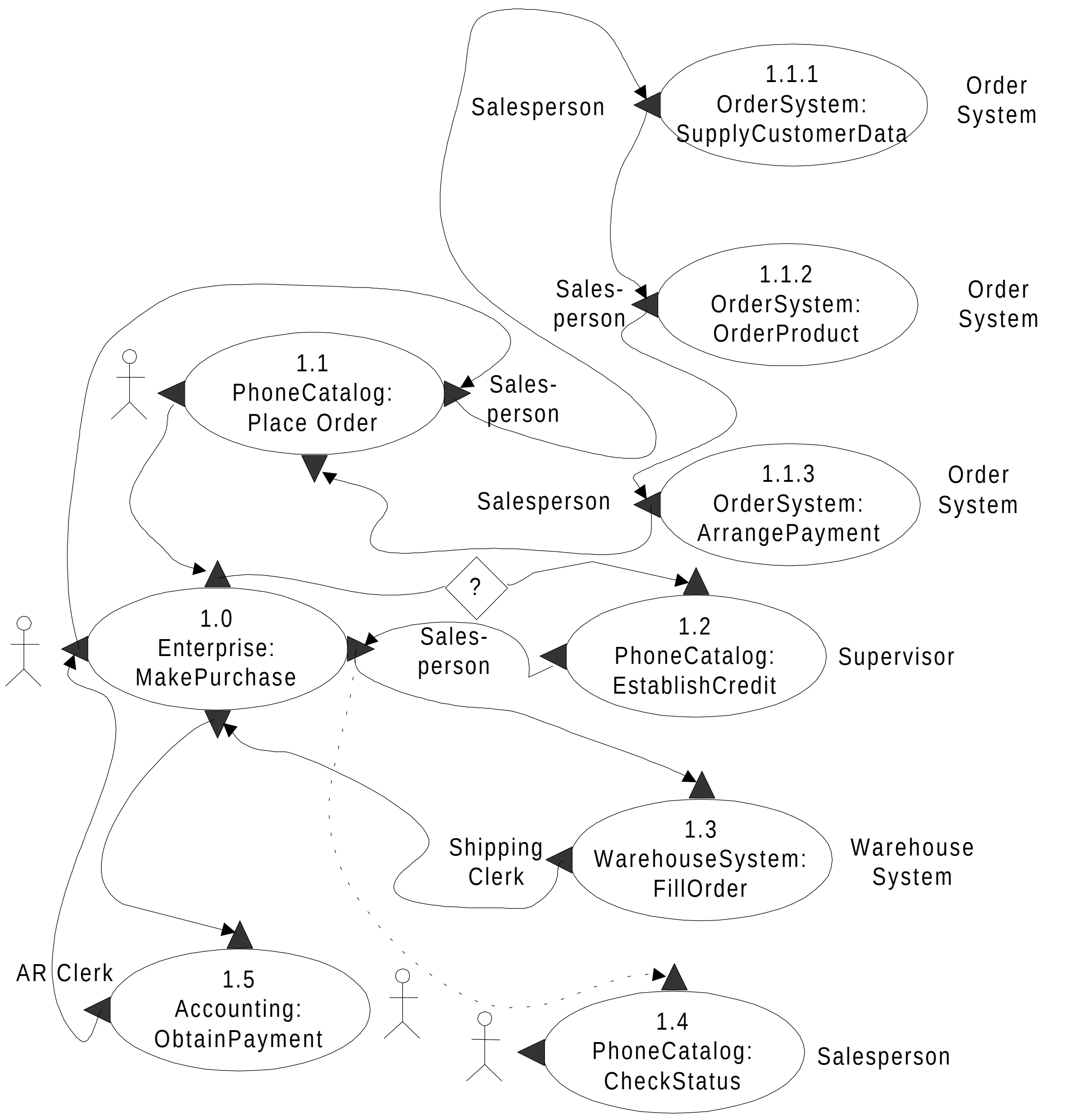

Salesperson
1.1.1 OrderSystem: SupplyCustomerData
Order System
Sales-person
1.1.2 OrderSystem: OrderProduct
Order System
1.1 PhoneCatalog: Place Order
Sales-person
Salesperson
1.1.3 OrderSystem: ArrangePayment
Order System
?
1.0 Enterprise: MakePurchase
Sales-person
1.2 PhoneCatalog: EstablishCredit
Supervisor
Shipping Clerk
1.3 WarehouseSystem: FillOrder
Warehouse System
AR Clerk
1.5 Accounting: ObtainPayment
1.4 PhoneCatalog: CheckStatus
Salesperson

Figure 5.16. Workflow Process Diagram with Modified ActionWorkflow Style Notation

Another presentation option is based on the ActionWorkflow™ notation. Adornments may be added to the ellipse use case symbol to represent states in the life cycle of the use case dialog and process flow. Each state is marked with a solid triangle attached to the use case ellipse similar to compass points. Process flows are presented as solid directed curved lines connecting the use case ellipses at the triangle points. Exception and extraneous process flows are represented as dashed directed curved lines. Parallel processes may be represented by multiple directed lines originating from a single triangle. Alternation may also be represented by a directed line to a special branching icon. The ellipse is used to represent a use case dialog between an actor and a subsystem. The actor always appears on the left end and the system on the right end. The actor symbol is only used for entities that are outside the top-most system being modeled.

<u>Mapping.</u> A use case dialog ellipse maps to an *AdaptiveUseCase* whose name matched the use case portion of the name string. The package part of the name maps to one of the stereotyped stakeholder models described in this document. Actor symbols map to *Actors* in the top-most system model. Solid line associations (all association in the modified ActionWorkflow notation option) map to *ClassifierRoles* at the originating point of the line.

5.7.13. Use Case Scenario Action Matrix and Use Case Dialog Map

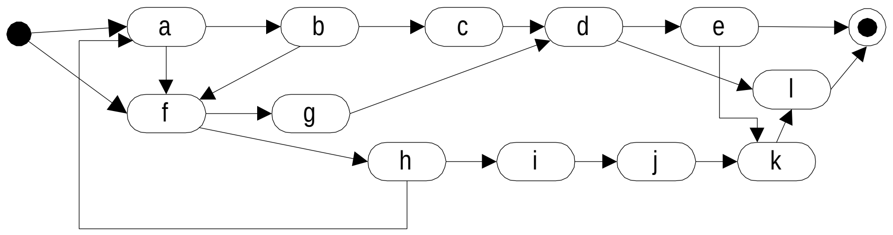
A use case scenario action matrix presents a cross-match between each use case action that is included in a use case with each scenario that performs the action. Specifically, a *UseCaseActionMatrix* is a stereotyped *ViewElement* that presents the complete set of *UseCaseActions* contained in an *AdaptiveUseCase*.

<u>Semantics.</u> Each scenario contains an ordered set of use case actions that describe its course of actions. The use case action matrix is intended to graphically present the scenario designated as the main course of action as a collection of cells that shows the coverage of its use case action by all variant scenarios. All additional use case actions not part of the main course of action are clustered separately.

<u>Notation.</u> Each cell corresponds to a unique use case action and scenario combination. The scenarios are represented as rows and the use case actions as columns. If a use case action is contained in the scenario, the cell is assigned one or more integers that correspond to the sequential ordering of the use case actions in the scenario. Since a single use case action may appear more than once in a scenario, the multiple integers permitted in each cell are separated by commas.

Presentation Options. An alternative presentation of the matrix is possible by converting use case actions into nodes in a directed graph. The sequential ordering provides the directed arcs that connect the nodes. Identification of the variant scenarios is lost, however through automated support, the path for each scenario can be highlighted. Additionally, all permutations that pass though two or more use case actions can also be used as a filtering mechanism to highlight paths. Of course, separate diagrams can also be generated for each filtering criteria. When the matrix is presented in this fashion, it may be alternatively referred to as a use case dialog map.

Mapping. Each column maps to a use case action. Each row maps to a scenario. The first row is always the scenario designated as the main course of action by the adaptive use case.



UseCaseAction	a	b	c	d	e	f	g	h	i	j	k	l
Main path	1	2	3	4	5							
Variant 1	1	2		5	6	3	4					
Variant 2	1					2		3	4	5	6	7
Variant 3	1	2	3	4								5
Variant 4	1	2	3	4	5						6	7
Variant 5	3	4	5	6	7	1		2				
Variant 6	3	4	5	6		1		2				7

Figure 5.17. Use Case Dialog Map and Action Matrix

5.7.14. Well-Formedness Rules

Stereotyped model elements are subject to certain constraints in addition to the constraints imposed on all elements of their kind. Each of the following stereotypes must conform to the following rules expressed in the Object Constraint Language (OCL) syntax of the UML.

AdaptiveUseCaseModel

[1] An actor must know what use case is invoked at its commencement (not necessarily which scenario).

```
self.contents->allReferencedElements (c | c.oclIsKindOf(UseCase)
   implies c.operation->forAll( o1, o2 | o1.name <> o2.name
      or not o1.parameter = o2.parameter))
```

<u>AdaptiveUseCasePackage</u>

[1] Each *AdaptiveUseCase* in an *AdaptiveUseCasePackage* must have a *mainCourseOfAction* of type *ArchetypeScenario* unless it adapts a *Scenario* contained in a different imported package.

```
self.contents->allReferencedElements (c |c.oclIsTypeOf(AdaptiveUseCase)
   implies ( c.mainCourseOfAction.oclIsTypeOf(ArchetypeScenario)
      or ( c.mainCourseOfAction.oclIsTypeOf(VariantScenario)
         and self.contents->exists ( c2 | c2.oclIsKindOf(Package)
            implies c2.contents->exists ( c3 | c3 =
               c3.mainCourseOfAction.requirement->select (
                  s | s.stereotype.name = ‘adapts’).supplier )))))
```

<u>UserUseCaseModel</u>

[1] A *UserUseCaseModel* must have exactly one association between any contained *UseCase* and an *Actor.* The *UseCase, Actor,* and *association* may be imported from another package.

```
self.contents->allReferencedElements (c | c.oclIsKindOf(UseCase)
      implies c.(allOppositeAssociationEnds->select (
      e | e.type.oclIsKindOf(Actor))).size = 1)
```

<u>ArchitectUseCaseModel</u>

[1] An *ArchitectUseCaseModel* contains exactly one customer use case package, plus a collaboration for each system level use case operation.

```
self.contents->select((c| c.oclIsTypeOf(CustomerUseCaseModel)).size = 1
   and c.contents->select( u | u.oclIsKindOf(UseCase)))->forAll ( u |
      u.operation->forAll(o: operation | o.collaboration <> Undefined))
```

<u>DesignerUseCaseModel</u>

[1] A *DesignerUseCaseModel* must not be a bottom-most level package.

```
self.contents->select (c | c.oclIsKindOf(Package)).size >= 1
```

<u>ImplementerUseCaseModel</u>

[1] No packages or models may exist that specify the subsystem or class being modeled.

```
self.contents->select (c | c.oclIsKindOf(Package)).size = 0
```

<u>AdaptiveUseCase</u>

[1] An *AdaptiveUseCase* must contain exactly one *Scenario* designated as the mainCourseOfAction.

```
self.scenario->exists( s | s = self.mainCourseOfAction)
```

<u>ArchetypeScenario</u>

[1] An *ArchetypeScenario* may not be based on any other scenario.

```
not self.contents->exists( s | s.oclIsTypeOf(VariantScenarioAction) )
```

[2] The *courseOfAction* for a *ArchetypeScenario* must contain equivalent sequences of *UseCaseActions* in its *useCasesegment*.

```
self.courseOfAction = self.useCaseSegment
```

<u>VariantScenario</u>

[1] Each *VariantScenarioAction* attached to a *VariantScenario* that has a *relativePlacementKind* value of *#replace* must have both a *startAction* and *endAction* contained in the *adaptedScenario's courseOfAction* from which the *VariantScenario* is derived. Furthermore the start action must either precede or be the same as the end action.

```
self.variantScenarioAction->forAll( v |v.relativePlacementKind=#replace
   implies ( self.adaptedScenario->exists (a | a = v.startAction)
      and self.adaptedScenario->exists (a | a = v.endAction)
      and (self.adaptedScenario->select ( a | a = v.startAction
         or a = v.endAction)->last = v.endAction ) ) )
```

[2] Each *VariantScenarioAction* attached to a *VariantScenario* that has a *relativePlacementKind* value of *#remove* must have both a *startAction* and *endAction* contained in the *adaptedScenario's courseOfAction* from which the *VariantScenario* is derived. Furthermore the start action must either precede or be the same as the end action.

```
self.variantScenarioAction->forAll( v | v.relativePlacementKind=#remove
   implies ( self.adaptedScenario->exists (a | a = v.startAction)
      and self.adaptedScenario->exists (a | a = v.endAction)
      and (self.adaptedScenario->select ( a | a = v.startAction
         or a = v.endAction)->last = v.endAction ) ) )
```

[3] Each *VariantScenarioAction* attached to a *VariantScenario* that has a *relativePlacementKind* value of *#inject* must have a *startAction* contained in the *adaptedScenario's courseOfAction* from which the *VariantScenario* is derived.

```
self.variantScenarioAction->forAll(v| v.relativePlacementKind = #inject
   implies  self.adaptedScenario->exists (a | a = v.startAction))
```

[4] Each *VariantScenarioAction* attached to a *VariantScenario* that has a *relativePlacementKind* value of *#regress* must have both a *startAction* and *endAction* contained in the *adaptedScenario's courseOfAction* from which the *VariantScenario* is derived. Furthermore the end action must precede the start action.

```
self.variantScenarioAction->forAll(v|v.relativePlacementKind = #regress
   implies ( self.adaptedScenario->exists (a | a = v.startAction)
      and self.adaptedScenario->exists (a | a = v.endAction)
      and v.endAction <> v.startAction
      and (self.adaptedScenario->select ( a | a = v.startAction
         or a = v.endAction)->first = v.endAction ) ) )
```

[5] A *VariantScenario* may contain at most one *VariantScenarioAction* with a *relativePlacement* of *prepend.*

```
self.variantScenarioAction->
      select(v | v.relativePlacementKind = #prepend)->size <= 1
```

[6] A *VariantScenario* may contain at most one *VariantScenarioAction* with a *relativePlacement* of *append*

```
self.variantScenarioAction->
      select(v | v.relativePlacementKind = #append)->size <= 1
```

[7] The range of all *UseCaseAction* reference points identified in *VariantScenarioAction* with a *relativePlacement* of *inject*, *replace*, *remove*, and *regress* must not overlap.

```
// produce a Set of subSequences
// produce a Set of qualifying variantScenarioActions
( (self.variantScenarioAction->forAll ( v |
      v.relativePlacementKind = #replace
      or v.relativePlacementKind = #remove
      or v.relativePlacementKind = #regress
      or v.relativePlacementKind = #inject )
   )->forAll ( v2 | self.adaptedScenario.courseOfAction->subSequence (
         v2.lowerRangePosition, v2.upperRangePosition) )
// since all OCL Collections of Collections are automatically flattened
// we can perform a Cartesian product comparison of all elements
)->forAll (e1, e2 | e1 <> e2)
```

[8] The *courseOfAction* for a *VariantScenario* must contain equivalent sequences of *UseCaseActions* in its adaptedScenario as modified by its variantScenarioAction list.

```
self.courseOfAction = self.variantScenarioAction->forAll (v |
   if    v.relativePlacementKind = #remove
   then self.adaptedScenario.courseOfAction
      ->remove(v.lowerRangePosition, v.upperRangePosition)
   else if v.relativePlacementKind = #replace
   then self.adaptedScenario.courseOfAction
      ->remove(v.lowerRangePosition, v.upperRangePosition)
         ->insert(v.lowerRangePosition, v.useCaseSegment)
   else if v.relativePlacementKind = #inject
   then self.adaptedScenario.courseOfAction
      ->insert(v.lowerRangePosition, v.useCaseSegment)
   else if v.relativePlacementKind = #prepend
   then self.adaptedScenario.courseOfAction
      ->prepend(v.lowerRangePosition, v.useCaseSegment)
   else if v.relativePlacementKind = #append
   then self.adaptedScenario.courseOfAction
      ->append(v.lowerRangePosition, v.useCaseSegment)
   else if v.relativePlacementKind = #regress
   then self.adaptedScenario.courseOfAction
      ->insert(v.upperRangePosition,
         v.useCaseSegment->union(self.adaptedScenario.courseOfAction
            ->subSequence(v.lowerRangePosition, v.upperRangePosition)))
   else endif endif endif endif endif )
```

<u>VariantScenario – Additional operations</u>

[1] The operation *position* results in an integer that identifies the *i-th* element of the *courseOfAction* sequence that is the same object as the operation's parameter.

```
position (object : OclAny) : Integer
position (object) = (Sequence{1..(self.variantScenarioAction->count)})
   ->select(index | self.adaptedScenario.courseOfAction->
      at(index) = object)->first
```

[2] The operation *lowerRangePosition* results in an integer that identifies the *i-th* element of the *courseOfAction* sequence that represents the lower bound of a subsequence identified by a *variantScenarioAction*.

```
lowerRangePosition (v : VariantScenarioAction) : Integer
lowerRangePosition = min ( self.position(v.startAction),
   (if self.position(v.endAction) = Undefined
   then self.position(v.startAction)
   else self.position(v.endAction)
   endif))
```

[3] The operation *upperRangePosition* results in an integer identifying the *i-th* element of *courseOfAction* sequence that represents the upper bound of a subsequence identified by a *variantScenarioAction*.

```
upperRangePosition (v : VariantScenarioAction) : Integer
upperRangePosition = max ( self.position(v.startAction),
   (if self.position(v.endAction) = Undefined
   then self.position(v.startAction)
   else self.position(v.endAction)
   endif))
```

[4] The operation *insert* on type *Sequence* results in the sequence consisting of the original elements of the sequence plus the additional elements contained in the second parameter's sequence inserted before the *i-th* element of the original sequence.

```
sequence->insert (index : Integer, iSeq : SequenceT) : Sequence(T)
(sequence->reject( sequence->forAll (e | at(e) >= index)))
   ->union( iSeq->union(sequence->subSequence( index, sequence->size)))
```

[5] The operation *remove* on type *Sequence* results in the sequence consisting of the original elements less the subsequence identified by the lower and upper *i-th* element parameters.

```
sequence->remove  (lower : Integer, upper : Integer) : Sequence(T)
sequence->reject( sequence->forAll (e |
   sequence->subSequence(lower, upper) ->exists( e2 | e = e2)))
```

<u>VariantScenarioAction</u>

[1] If *relativePlacement* is *#replace*, then both a *startAction* and an *endAction* must be designated.

```
self.relativePlacementKind = #replace implies (
      not self.startAction = Undefined
      and not self.endAction = Undefined)
```

[2] If *relativePlacement* is *#remove*, then both a *startAction* and an *endAction* must be designated.

```
self.relativePlacementKind = #remove implies (
      not self.startAction = Undefined
      and not self.endAction = Undefined)
```

[3] If *relativePlacement* is *#inject*, then only a *startAction* should be designated.

```
self.relativePlacementKind = #inject implies (
      not self.startAction = Undefined and self.endAction = Undefined)
```

[4] If *relativePlacement* is *#prepend*, then neither a *startAction* and an *endAction* should be designated.

```
self.relativePlacementKind = #prepend implies (
      self.startAction = Undefined and self.endAction = Undefined)
```

[5] If *relativePlacement* is *#append*, then neither a *startAction* and an *endAction* should be designated.

```
self.relativePlacementKind = #append implies (
      self.startAction = Undefined and self.endAction = Undefined)
```

[6] If *relativePlacement* is *#regress*, then both a *startAction* and an *endAction* must be designated.

```
self.relativePlacementKind = #regress implies (
      not self.startAction = Undefined
      and not self.endAction = Undefined)
```

<u>RelativePlacementKind</u>

[1] The domain for *RelativePlacementKind* is *replace, remove, inject, prepend, append,* and *regress*.

```
RelativePlacementKind =
   enum { replace, remove, inject, prepend, append, regress }
```

<u>UseCaseActionSequence</u>

[1] A *UseCaseActionSequence* may only contain specializations of *UseCaseSegments*.

```
self.allContents->forAll (c | c.oclIsKindOf(UseCaseSegment)
      and not c.oclIsTypeOf(UseCaseSegment))
```

[2] All leaf nodes of a *UseCaseActionSequence* composite hierarchy must be *UseCaseActions*.

```
self.allContents( c | c.oclIsTypeOf(UseCaseActionSequence)
      implies c->exists (c2 | c2.useCaseSegment->forAll (
            c | c.oclIsTypeOf(UseCaseAction) ) ) )
```

<u>Adapts</u>

[1] An «adapts» stereotyped dependency relationship between a VariantScenario as the source and another Scenario as the target must contain at least one *VariantScenarioAction* element.

```
self.allConnections->exists (e | e.type.oclIsTypeOf(VariantScenario)
      implies self.allConnections->size = 2
            and exists (e | e.type.oclIsTypeOf(Scenario)
            and e.type.variantScenarioAction->size >= 1 ))
```

WorkflowProcessDiagram

[1] A *WorkflowProcessDiagram* can only be used to represent a single *ArchitectUseCaseModel*, *DesignerUseCaseModel*, or *ImplementerUseCaseModel*.

```
self.model->select ().size = 1
      and self.model->forAll (c | c.oclIsKindOf(ArchitectUseCaseModel)
      or c.oclIsKindOf(DesignerUseCaseModel)
      or c.oclIsKindOf(ImplementerUseCaseModel))
```

5.8. Discussion

The static semantics of UML define how an instance of a construct should be connected to other instances to be meaningful. The OCL is used to express these static semantics. The constraints take the form of invariants that must be satisfied for the construct to be meaningful. Additional operations may also be defined for the OCL expressions. Although not required, these additional operations clarify the purpose of the constraint by substituting a complex expression with a meaningful operation name. Evaluation of an OCL expression with an additional operation defined is performed by what effectively amounts to a macro substitution. Through elaboration of OCL constraints on adaptive use case model elements, the model presented in this chapter has a more formal basis than the use case model elements defined in the UML.

The UML Semantics documents states that each method performed by a use case instance is performed as an atomic transaction, meaning that it cannot be interrupted by any other use case instance. This statement has two implications for use case modelers. First, transaction management must be considered when creating methods that invoke other subsystems or components, since the transaction management schemes for the components will be unknown to the calling method that realizes the use case operation. Second, for traceability purposes a one to one mapping between operations and methods should exist. The adaptive use case mechanism provides for this direct mapping between operations and methods, as shown by the «derived» stereotype mapping between a use case action and the UML *ActionSequence* model element in Figure 5.9. Since each use case action can be adapted through the variant scenario element, there is an implicit extension point associated with each use case action. Thus, adaptive use cases avoid having to define every action explicitly as an extension point in order to anticipate any future variability. This capability overcomes the weaknesses inherent in the «uses» and «extends» stereotypes for use cases. Moreover, explicitly mapped actions provides greater precision for traceability purposes.

The UML allows for attributes to be attached to describe functionality. Potential attributes may include a glossary of terms, special requirements, supplemental specifications, preconditions, scope, priority, frequency, importance, and goals. Although these may be important facets of a use case, they remain informal descriptions. Business rule mechanism can provide more formalization for many of these. For example, a glossary of terms may be stated as a type of business rule structural assertion. Preconditions are can be formalized as behavioral assertions. Furthermore, many of these may be derived from the formalized representations in adaptive use cases. Chapter six exploits this synergy between business rules and adaptive use cases by demonstrating how adaptive use case model elements interact with business rule declarations. Chapter seven then provides a context for the integration of these two mechanism (adaptive use cases and parameterized business rules) through the domain evolution architectural transformation (DEAT) process model.

The «adapts» dependency stereotype relaxes the concept of strict inheritance. The only UML reference to strict inheritance versus non-strict inheritance is in the description of State Machines. Because of this, adaptive use cases provide a more flexible mechanism than UML state machines or the business process oriented activity models that are specialized subtypes of state machines. Since the use case actions are precisely defined, creating sequence diagrams or activity models is a straightforward mapping. Thus, automated generation of sequence diagrams is possible with adaptive use cases. This is not possible with the less formal UML use case constructs.

The UML strict inheritance policy restricts the degrees of freedom for variability mechanisms. By adopting and modifying frame technology [Bass97] with a more formalized use case organization, adaptive use cases relax this restriction. The adaptive use case model provides a variability mechanism in a manner that is specifically tailored for describing use cases. Additionally, by integrating the speech act model [WF87] into the adaptive use case model, a more robust representation of use case is provided than that found in Cockburn’s success/failure dichotomy for use case goals [Cock97].

The adaptive use case model elements differs significantly from Basset's frame processor language. The Basset's frame technology utilizes five basic building blocks: frame delimiters, adaptation commands, variable assignments, case statements, and while loops. For our purposes, the frame delimiters are at discrete levels of granularity, i.e. the *UseCaseAction*, *UseCaseSegment*, *Scenario*, and *AdaptiveUseCase*. Adaptation commands are built into the *VariantScenarioAction* model element. The remaining three building blocks relate to adaptive use case parameter bindings. Thus, this approach provides all of the capabilities of adaptive frame technology through model elements, their associated well-formedness rules, and notational representations as view elements. In essence, adaptive use cases define a domain specific language for frame technology. The domain being referred to here is really a meta-domain, representing the management of a business domain architecture. The actual domain language for a family of applications, such as school administration, human resource management, or forms management, is captured in the business rules that are attached to the use cases. How this is accomplished will be presented in chapter six.

The adaptive use case mechanism is intended allow dynamic modifications to use case scenarios that are contained within the same model management package. Thus, this capability integrates the dynamic features of use case maps [Buhr97] with the model management features found in [JGJ97]. The «configuration item» and «configuration» stereotypes from [JGJ97] are semantically consistent with adaptive use cases. However, since the adaptive use case model does not impose a build time binding on the components that realize the use cases, self modifying dynamic properties are possible. Chapter six further elaborates on this capability through the *Self-Modification* business rule pattern.

Adaptive use cases combine the best features from object-oriented development methodologies and the workflow process approaches. One significant example of this is the development of the use case dialog map and action matrix. As pointed out in the discussion of Horn's information mapping techniques [Horn89], no two domains are exactly alike. Thus, these additional representation devices provide versatile tools for visualizing and understanding the domain model that may be added to the arsenal of domain model tools and techniques. These tools exploit the use case refinement level model developed in this chapter which enables view focus to be tailored to various stakeholders. This model provides a context for viewing higher level use case actions by lower level stakeholders. Thus, it provides more flexibility than any other use case modeling approach. Use case dialog maps provide a new graphical representation of use cases that facilitate iterative transformations as prescribed in the DEAT process model from chapter seven.

CHAPTER VI

BUSINESS RULE MODELS, PARAMETERS, AND PATTERNS

This chapter formally defines a business rule meta-model that is complementary with the adaptive use case model presented in the previous chapter. The Unified Modeling Language (UML) notation is once again used to present the structural and dynamic relationships to these model elements. Specifically, this chapter first defines business rules and establishes a context for their use. Next, it describes a business rule meta-model in terms of types, associations, attributes, grammar, and states. Having established the conceptual foundation for the business rule model, this chapter proceeds with development of a business rule pattern language that follows the rule types previously established. The chapter concludes with the introduction of business rule templates and demonstrates how they carry forward the theme of template bindings introduced with adaptive use case templates.

6.1. Introduction

Business rules are the drivers for specifying and configuring an application. Business rule models capture and segregate policy decisions from the functional processes captured in use cases. They assert the necessary data structures, while use cases declare the necessary behavioral semantics. In turn, use cases drive the selection of components and classes. The separation of a domain model into policy, process, and structural views provides greater flexibility and more understandable representations.

This thesis approaches the integration of business rules and use cases into a domain model from a policy-oriented viewpoint. The business rule model provides a complementary perspective to the process-oriented use case model and the structure-oriented class model. Workflow oriented rules are primarily used for process flow control. Such rules determine which scenario of a use case is instantiated. Data oriented rules are primarily used to establish necessary entity structures and associations. By introducing business rule bindings to use cases, these static object and data structures of an application can also be controlled. However, unlike a class model, which directly integrates rules as constraints and invariants, these business rules are bound to process definitions and are only indirectly related to classes through the realization of use cases. Business rule binding allow process oriented speech acts to highlight the different contexts of system boundaries that isolate structural differences. They also enable structurally oriented adaptive frames to be used to isolate behavioral differences.

Business rules utilize use case templates to create sets of scenarios that comprise the use case. These business rules are not directly integrated into the scenario; rather they are bound at build time. In this manner, several different parameterized data structure formats can be accommodated in a single use case. For example, in a use case called “OpenNewAccount” for a banking application, an individual, a corporation, or a trustee may play the role of Actor. The information required by each of these will vary to some degree. However, the realization of the use case will require coupling across the collaborations of cooperating objects in order to consistently deal with the variety of data structures accompanying the different players of the actor role. Parameterization in the form of the business rule allows a single parameter to declare the mutual constraints that define this coupling.

It should be emphasized that business rules are primarily concerned with application specific data. Rarely should parameterization of workflow related data, such as process or activity state, be necessary. Although business rules govern the transition from one workflow state to another, they are generally consistent across all scenarios for a use case. A difference in the workflow will usually manifest itself as a new scenario. This new scenario will still utilize the same process and activity states.

The binding of business rules to use cases represents a novel approach that sets it apart from previous approaches. There has been considerable discussion with respect to business rules that *suggest* a relationship between use cases and business rules. Based on the comprehensive research regarding use cases as described in chapter two, the approach described in this thesis is the first to formalize the relationship through modeling constructs.

When viewed from an analysis perspective, one line of research has investigated the nature of rule constructs. Such work represents a micro view. Each rule is decomposed into component pieces with accompanying graphical notation, such as constraints and attributes [Ross97], [Silv95]. Other work has treated rules in a behavioral context, such as collaborations role stereotypes [Chaf96] and event-condition-action statements [Herb95]. These stereotyped building blocks represent an intermediate view of rules. The approach here takes a macro-view that deals with the dependencies among the assertions and derivations that relate to processes transcending individual collaborations`.

From a design perspective, object-oriented development and most other system development methodologies subsume business rules directly into the objects [Gott97] and model notations. A slightly different approach separates rule inferencing from the objects by specifying an object/inference-engine interface [ILOG97]. This interface requires the addition of a number of member functions associated with certain classes so that they can be accessed as if they were defined as assertions, however no additional data members are required [Fran90]. A third approach utilized dynamic reflection and a meta-object protocol. This meta-layer contains a set of default rules about how the object system works, how methods are added, and how classes inherit from superclasses. Default rules can be modified, providing a uniform semantic and syntactic access to objects that are conceptually the same but which have different implementations [Fran97].

The parameterized business rule binding approach has many similarities with dynamic reflection. First, one of the key features is maintaining knowledge about dependencies among elements so that when changes are made to various objects, these changes can be automatically propagated throughout the entire system. Second, functionality is abstracted and embedded into the system. Third, an application is build as a series of layers, with the topmost layer being a use-friendly scripting language that represents domain-specific concepts and application specific rules.

The primary difference between parameterized business rule binding and dynamic reflection through meta-object protocols relates to the emphasis of analysis in the former and design in the latter. Dynamic reflection tracks dependencies of classes and subclasses. Parameterized rules track dependencies of use cases, which may be realized by many different object systems and therefore is more generic. Dynamic reflection assumes a run-time flexibility that imposes a certain architecture. Despite lazy evaluation strategies, it also assumes certain performance costs for this flexibility. Parameterized rules make no such assumptions, but also does not rule out this implementation. Through build time binding instead of run time binding, parameterized rules can be realized through alternative architectures. This approach depends on the existence of fundamental generic components that realize the domain model in a manner that is not committed to a specific data structure until build time, such as the Demeter method [Lieb96].

Rules are related through membership in rule sets. These sets also share the adaptive frame technology semantics. A web of related rules are combined into a rule set as the maximal coverage of a root rule that references other rules as an antecedent of consequence. Adaptive frames may override individual rules or complete rule sets. Moreover, rules may be defined for selection of adaptive frames. Other approaches have used rule sets, but not in this same context. The Smart Object Language (SOL) approach has been applied to workflow management systems utilizing reflection in order to make appropriate decisions about routing, triggering and monitoring work [VJK96]. The use of rule sets is centered on control abstraction rather than functional behavior. Domain knowledge is embedded in the methods section of smart objects as a set of if-then rules. A *monitor* section contains the control logic, also as if-then rules. The *attributes* section allows different rules to be applicable at different locations, but these only apply to control states and not data structures. Silva has proposed a nested rule model that provides a multi-level representation of the declarative and active behavior through rule sets, coupling modes, and interactions [Silv95]. Graham creates rulesets as a new member type in objects along with attributes and operations. All of the approaches address interrelationships between operations and attributes only within the object. For example, business rules that declare structural assertions typically relate one attribute to another attribute. Triggers, which are a form of an action assertion, relate operations to attributes. Exception handling can be composed of any combination of one or more attributes and operations.

Since all of these described approaches are object based, inter-relationships among objects must be embedded in another object. Global rule objects could overcome this limitation, but has a side effect of tighter coupling between components and classes. The approach of binding business rules to process models provides a higher degree of flexibility in the specification of the design. This is accomplished without the need for extensions to the implementation language.

Business rules assert the necessary data structures, while use cases declare the necessary behavioral semantics. Thus, use cases drive the selection of components, classes, or manual processing through domain asset fit assessment and project resource constraints. By recognizing recurring types of business rule declarations, entire rule sets may be specified by identifying a business rule pattern and associating a set of parameters with the pattern. Once mappings have been established between the business rule pattern and the use case, modifications to business rules may be automatically propagated to the use cases and ultimately to the subsystems and classes that realize the use cases.

6.2. Business Rule Meta-Model

6.2.1. Business Rule Types

Parameterized business rule patterns are classified into eight categories. The first four are primarily concerned with structural declarations. Attribute patterns focus on a validation of the state of a single attribute. Collection patterns address the relationship of a population of objects. Collections are typically composed of similar types that share common characteristics. The relationship pattern family deals with associations between objects. Whereas collection patterns deal with similarities, relationship patterns are primarily concerned with differences that can be expressed through logical conjunction and disjunction expressions. The key distinguishing feature of temporal business rule patterns is the perspective of time that can constrain existence or maintain version history.

The remaining four pattern families are primarily concerned with behavioral declarations. Transaction patterns deal with events, states, and units of work. Control patterns address workflow related concepts. Derivation patterns focus on algorithmic constraints. Finally, the protocol family of patterns specifies contexts with respect to the domain model itself.

This classification into eight business rule pattern categories may be compared with several other classification schemes that have been proposed. A comparison of five such approaches is made in Table 6.1. The Unified Modeling Language [UML97] established a core meta-model that can be used to model many of the patterns. However, since is does not have strong support for business process modeling the behavioral declaration types are not well supported. This is particularly true for the control family of patterns. The GUIDE Business Rule Meta-Model [HH95] addresses both structural and behavioral types. Its primary area of deficiency is support for collections. The Active Object-oriented Database Modeling Technique proposed by Silva [Silv95] provides consistent support for each type of pattern except for derivations. The Ross method [Ross97] is the most comprehensive of any of the approaches compared. Its primary deficiency lies in process aspects, due to its fundamental restriction of only considering persistent data as a basis for rules. Graham’s Semantic Object Modeling Approach (SOMA) method, which has been integrated into the Open Modeling Language, is the least comprehensive of the approaches compared here. Under SOMA, rules are directly bound to classes. Although global rules may be defined between objects, no direct support is provided.

Table 6.1. Comparison of the Parameterized Business Rule Pattern Classification to Other Approaches

Patterns/Approach →	UML	GUIDE	Silva	Ross	SOMA	Comments
Attribute						
Range Set	✶	✶	✔	✔	✔	e.g. age between 21 and 65
Interval	✶	✶	✔	✔	✶	Increment-by, rounding
Domain	✔	✶	✔	✔	✔	Enumeration types, min/max
Type membership	✶					Include conversion between types
Default Value	✔	✶		✔	✶	initial value if null
Collection						
Multiplicity	✔		✔	✔	✶	Mandatory, limits, and cardinality
Ordering	✔			✔	✶	key function definition
Uniqueness	✶	✶	✔	✔	✶	relates to values, not object identity
Recursion	✶		✶	✔		e.g. composite pattern
Aggregation	✔		✶	✔		e.g. frequency distribution of values
Relationship						
Sub-typing	✔	✔	✶	✔	✔	includes disjunction and overlapping
Association	✔	✔	✔	✔	✔	and, or, if exists, for all
Temporal						
Association	✶	✶	✶	✔		relates creation to others' existence
Entity State History	✶	✶	✶	✔		functional versioning of attribute
Aggregation	✶			✔		rates, counts
Transaction						
Unit Of Work	✔	✔	✔	✔		atomic transaction definition
State Transitions	✔	✔	✔		✶	progressive, reinitiating, cycle, etc
Triggers	✔	✔	✔	✔	✔	when needed, when changed
Event Management	✶	✔	✔	✔		mutual dependencies, synchronized
Deferred Action	✔	✶	✔	✔		queuing, blocking, etc.
Control						
Priority	✶			✔	✔	conflict resolution
Schedule Execution	✶	✔	✶	✔		absolute, relative executions
Role		✶		✶		role, organization structure mapping
Authorization		✔	✶	✶	✶	creation, override, execution rights
Contingency	✶	✶	✔	✔		timed window for execution
Enforcement		✔		✔	✶	prevent, notify, log, permit
Self-Modification						reconfigure, recompile components
Derivation						
Algorithm	✔	✔		✔		mathematical calculations
Inferred Fact		✔		✔		inferencing, induction, deduction
Type Conversion						Interface
Type Migration						schema evolution
Interpolation						specialized algorithm
Protocol						
Contract	✔	✶	✶			arguments, types, order
Alternatives	✶	✶				alternative, extension actions
Exceptions	✶	✶	✶	✶	✔	conditions for exception actions
Explanation		✶	✶	✔		Operation, rule invocations

Legend: ✔ = explicit or strong support; ✶ = implicit or sufficient support;
= unintuitive support – either requires compound constructs to support, or is unsupported

6.2.2. Business Rule Associations

Unlike the other approaches compared in this chapter, business rules are not attached to other entities. Rather they are first class analysis modeling elements. The use of business rule patterns allows a business rule to be parameterized and participate in relationships with other modeling elements. The exception to this is the classification of business rules that prescribe requirements for the attributes of entities. We raise this distinction here so that one will not confuse attributes of a business rule with business rules that specify attributes.

A business rule is defined in the context of its association to other modeling elements, as shown in Figure 6.1. In particular, a business rule directly links to conditions and actions contained in use case scenarios. Events are the third primary link to the process oriented use cases and the structure oriented class models. Events may be raised as the result of a use case action or as the result of a real world entity that exists outside the system boundary. Such real world events may be attributed to the passage of time or as the result of instrument sensors. When new events are raised as the result of a use case action, a chaining of business rules occurs. If the process is fully automated, this chaining may continue until the process runs to completions.

Two other entities are associated with a business rule. The first entity is a calendar. This enables a business rule to address temporal aspects, such as schedules. The other entity is a position, or role within an organizational structure. The association between a business rule and a position is a two-fold one. The first established the authority for the rule. Depending on the placement of the rule within the organizational structure, a rule may be overridden by a rule placed at a higher level. The rule's proponent identifies who is responsible for maintenance of the rule. These two associations may identify the same position, but this is not required.

6.2.3. Business Rule Attributes

A business rule also contains four attributes. These are the strength, scope, source, and state of the business rule. The strength of the business rule refers to the enforceability of the rule. Scope refers to its application in use cases. A rule with global scope must be applied to all use cases and all organization units, if applicable. Alternatively, a rule may be restricted to certain workflow participants and use cases. The source of the rule relates to the origin of the rule. A rule may originate from legislative change, market demands, industry practice, or even imaginative users of the system. There are five primary process states of a business rule: collected, validated, classified, active, and inactive. Once a rule is created, it may be incomplete in one or mores respects. If a business rule is not well formed, it is in the collection phase. A business rule must be mapped to use case and subsystem elements in terms of events, conditions, and actions. Once this has been completed, the rule is validated. When a rule is completed in terms of complete specification of attributes and remaining associations, it is complete, or classified. Once the rule is implemented, it becomes either active or inactive depending on the context of its use in a particular application.

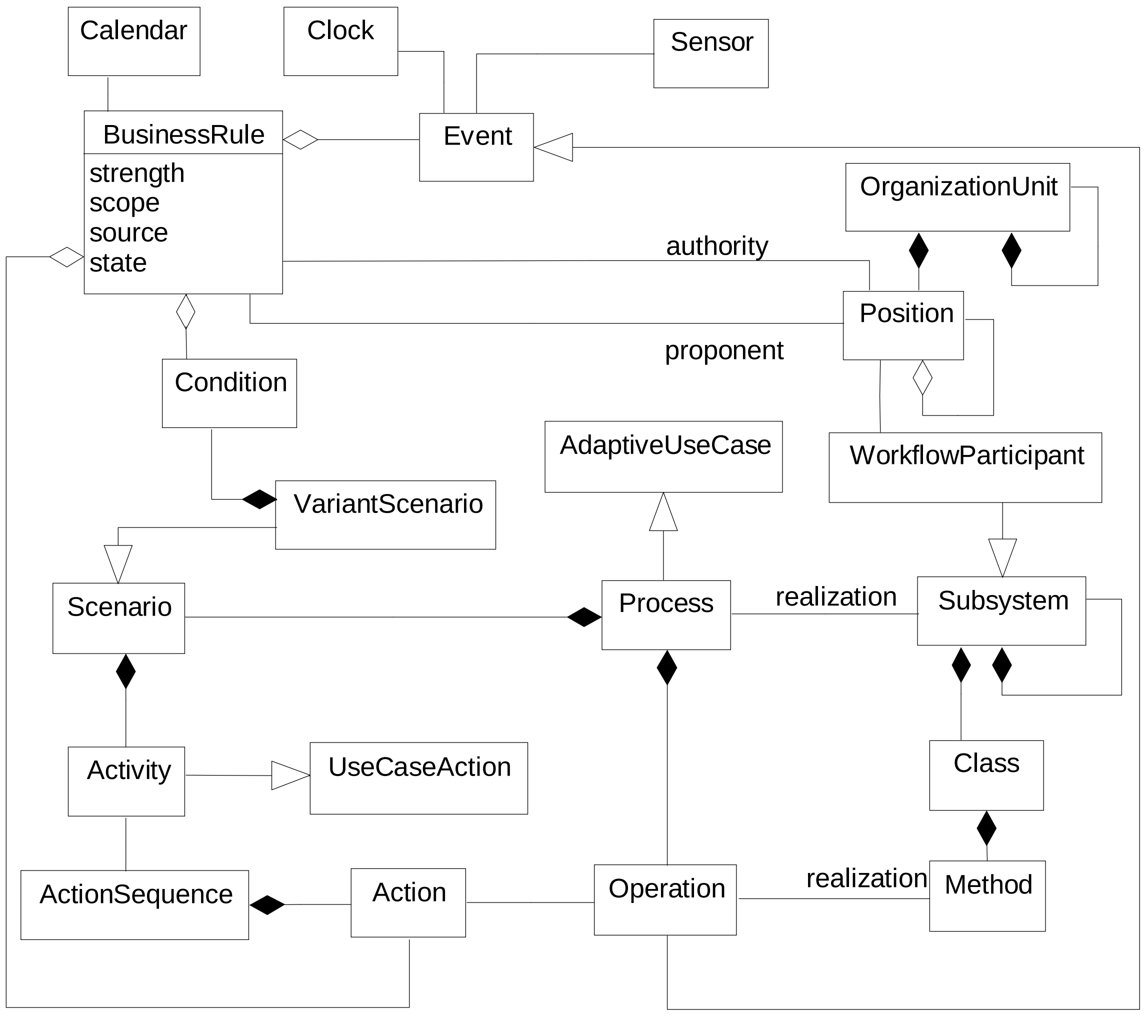


Figure 6.1. Business Rule Meta-Model

6.2.4. Business Rule Grammar

Business rules can invoke actions in response to events upon successful evaluation of conditions. Thus, conditions play an important part in determining the course of action that will be taken when performing a use case. Fortunately, linking the conditions in the business rule model to the *VariantScenarioAction.condition* attributes in the use case model is straightforward. However, the sentence structure of use case action statements should be constructed in such a way that they are compatible with the sentence structure for business rules. The business rule syntax presented here synthesizes elements from use case action statement grammar [Grah95][Cock97], text ordering [HH95], and event-condition-action declarations [Herb95], [Silv95] with business rule parameter sets.

Business rules have multiple ways of being stated. The relationship between a set of business terms and business rule declarations can be expressed through multiple semantically equivalent text orderings. For example, a binary association between business terms can be expressed in two ways. The following two statements represent the same rule:

- An employee must have a job description.
- A job description must be assigned to each employee.

The business rule model must normalize such rules so that they are only represented as a model element once, although they may be presented in several ways. The convention specified in this model is to follow an event-condition-action sequence to text ordering as shown in Table 6.2.

Table 6.2. Business Rule Grammar Components.

Expression	Description
Business rule	[event], condition, action
Event	(when needed | when changed | when scheduled | when action completed)
Condition	if [not] (authorization | constraint | logical expression)
Authorization	[only] x may do y
Constraint	[each] x must (be | have | do) y
Action	Actor, verb, subject, target
Actor	Use case actor
Subject	Entity that moves, changes, or is created
Target	Entity that receives or responds to subject

Syntax: “[ ]” = optional; “(|)” = a choice

Consider the following statement: “Every Friday, if an item is more than three weeks overdue, the librarian sends an overdue notice to the lender.” The event clause (every Friday) is a “when scheduled” type of event. The condition clause is a logical expression. The action clauses consists of an actor (librarian), a verb (sends), a subject (an overdue notice) and a target (to the lender).

6.2.5. Business Rule Execution State

In addition to the process state that is defined as a meta-attribute for a business rule, an instance state attribute also may be logically defined. Five business rule execution states may be derived through three Boolean attributes – *isTrue*, *isEnabled*, and *isInvoked*. If the rule is not enabled, then the other two attributes are irrelevant. In this case, the execution state is null, or *unknown*. If the rule is enabled, then the rule may be either invoked or not. The execution state is *invoked* if the *isInvoked* attribute is true, but the *isTrue* attribute is null. When a rule is currently invoked, it is in an atomic action state that will either result in an execution state of *true* or *false* if successful. Otherwise, the rule firing caused an *exception* state.

The representation of execution state is mainly an implementation issue. However, some processes may have complex chaining of conditional expression evaluations that occur over an extended period of time. As a result some rules may need to have refer to these execution states.

6.3. Parameterized Business Rule Patterns

Business rule patterns recognize common domain analysis problems that can be expressed through parameterized modeling constructs. Therefore, the focus of the business rule pattern language is specification rather than problem description aspects of analysis patterns [Fowl96], the solution aspect of architectural [BMRS96] and design patterns [GHJV94], or the project management aspects of process patterns [Cope94], [Beed97b], as shown in Table 6.3. The basic premise behind business rule patterns is to provide powerful ways to express requirements. By identifying recurring domain analysis modeling constructs, these patterns enable a solution to be generated from reusable components. Only a minimal set of parameters is required to configure an application’s behavior. The key to making this work is separation of the expression of the requirements in the problem domain with the actual implementation code in the solution domain. This requires a mapping of the business rule patterns to compatible components. However the actual selection of components is not dependent upon the method in which the requirements are expressed since transformations of concise representations to the most generic case are provided.

Table 6.3. Characterization of Pattern Types

Pattern type	Focus
Business rule	Problem specification
Analysis	Problem description
Architectural	Problem solution – architecture
Design	Problem solution – design
Idiom	Problem solution – implementation
Process	Project management and organization

Each of the business rule patterns presented in this section follow a common format for expressing patterns. First a problem and its context are described that establish the motivation for defining a business rule pattern. The problem description often reveals a system of forces that arise in the context. Then a solution is presented that describes the participants of the business rule pattern and how they collaborate to solve the problem. Through the solution description and subsequent examples, guidance is given for when and how to implement the pattern, possible variants, strengths and weakness. If applicable, a graphical representation of the pattern's structure in UML notation is provided along with special mappings to UML model elements. In general, the business rule pattern categories are mapped to UML model elements and diagrams as shown in Table 6.4. Finally, reference to any related patterns is described.

As previously shown in Table 6.1, the pattern language presented in this section may be grouped into eight categories. A general affinity exists between member business rule patterns of the same category that encourages their combination for representing complex business rules. These categories are not intended to imply incompatibility between composition of patterns from several different groups. The examples provided in this section demonstrate many such synergies.

The first few patterns presented are described in greater detail than the remaining ones. The intention is to elaborate each of the conceptual facets of the business rule pattern in depth when they are first introduced. First, more discussion is provided in the problem and solution parts by introducing general examples. Second, cases are provided that elaborate actual instantiation of the defined parameter. Finally, specialized shorthand representation of more generic parameter sets is described. Although elaboration of case attribute values is omitted in most subsequent patterns, they should be evident from the discussion of each pattern without further elaboration. Additional specialized parameter sets of business rule patterns are readily conceivable that may be appropriate to other business domains.

Table 6.4. Relationship between Business Rule Patterns and UML Constructs

Rule pattern	UML model element	UML diagram
Attribute	DataType	Static Structure Diagram
Collection	Type	Static Structure Diagram
Relationship	Collaboration	Collaboration Diagram
Temporal	Type	(None)
Transaction	StateMachine	Statechart Diagram
Control	UseCase	Activity Diagram
Control – Meta-Level	Component	Implementation Diagram
Control – Authority	Actor	Use Case Diagram
Derivation	Expression	(Uninterpreted)
Protocol	Interface	Sequence Diagram

6.3.1. Rule Pattern: Range Set

Problem – Describing an Extent between Two Points. Processes often involve conditional branching of activities based on certain attributes. Frequently, thresholds for an attribute are established in order to define discrete states that dictate which activities are applicable. The process must know the various thresholds in order to determine the correct state. This involves describing an extent, or distance between two points that utilized some type of measurement.

Solution. The range set pattern defines a range extent type and a unit of measure. It also names each member of the range extent type along with quantifying the upper and lower bounds. The range extent type becomes the state attribute that contributes to identifying the correct conditional behavior. The lower and upper bounds are expressed in terms of the unit of measure.

The unit of measure identifies the scale of the ranges to insure that all measurements are comparable. This may require a *Derivation* rule to convert from one unit of measure to another. Depending on whether or not the measurements themselves are discrete or continuous, additional specification parameters may allow an overall reduction in the volume of data needed to capture the same information.

Example – Course Grades. This example describes a grading scale utilizing the *Attribute Range Set* rule pattern. Several parameters are introduces to fully specify the range.

Range Extent Type:	< Range Extent Type>
Unit of Measure:	<Unit of Measure Type>
Complete:	Yes | No
Open Ended:	Neither | Both | Upper | Lower
Range Extent Type:	Grade
Unit of Measure:	Percentage of correct answers
Complete:	Yes
Open Ended:	Neither

	Lower Bound		Upper Bound	
Grade	ComparativeEvaluator	Value	ComparativeEvaluator	Value
A	≥	90%	≤	100%
B	≥	80%	≤	89%
C	≥	70%	≤	79%
F	≥	0	≤	69%

Example 6.1. Course Grades Parameters and Case Values

If we make a further assumption that the range is complete for all measurement, then further simplification is possible. By declaring a measurement type parameter as complete, it is possible to declare a single meta-attribute of the rule pattern that specifies the comparative evaluator for each value. Thus, the number of parameters is reduced from (4 + 5r) to (4 + 2r), where (r) is the number of ranges.

Parameters:
Measurement Type: Continuous
Inclusive: Lower

Grade	Threshold
A	90%
B	80%
C	70%
F	0%

Example 6.2. Modified Course Grades Parameters

UML Representation and Related Patterns. The *Range Set* business rule pattern consists of specification and realization model elements. The two examples in the section (Example 6.1 and Example

6.2) are shown as specializations of the *RangeSpecification* type. The *continuousMeasurementKind* is a named specialization type of *RangeSpecification* and is itself a parameter that must be specified that includes an enumerated type domain of all named specializations of *RangeSpecification.* When no measurement kind is specified, the generic case applies. The realization model elements must then be fully described to complete the modeling of the business rule.

The *Range Set* business rule pattern shares concepts with Fowler's *Range* pattern [Fowl96], but here the focus is primarily with specification aspects rather than descriptive ones. The *Derivation* business rule pattern can be applied in collaboration with the *Range Set* pattern to allow conversion from one unit of measure to another, if required. The *Domain* business rule pattern defines the domains for each enumerated attributed.

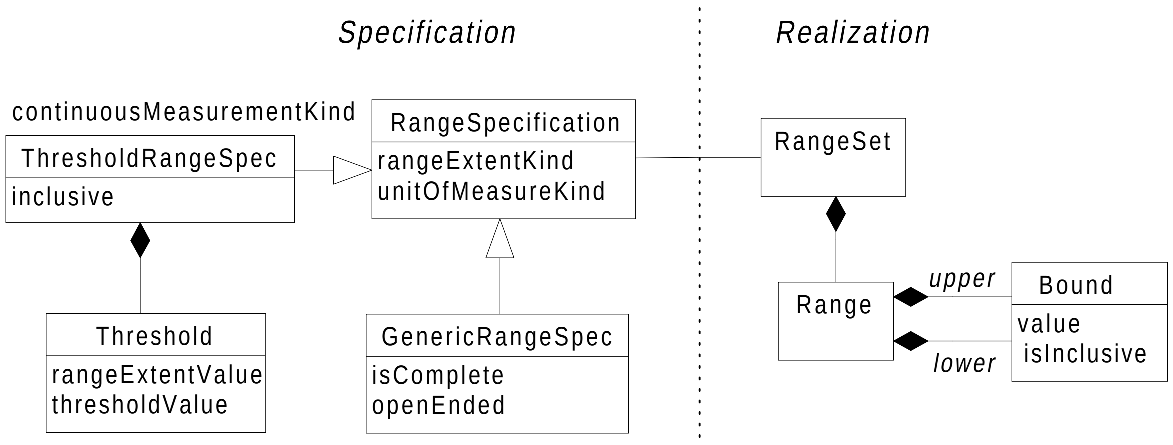


Figure 6.2. UML Representation of Attribute Range Set Pattern

6.3.2. Rule Pattern: Interval

Problem – Quantitizing Values. An attribute's value may be assigned either as the result of a measurement, a calculation, or both. Performing addition, subtraction, multiplication, and division operations may yield meaningless results depending of the type of phenomenon being measured. This is particularly true for performing arithmetic operations involving space, time, or currency. Measuring the time between two dates results in a different unit of measure, such as days. Furthermore, calculations between two points in time may be subject to business rule driven approximation. For example, a business week may define the measurement between 9:00 AM on Monday, February 2, 1998 and 5:00 PM on Friday, February 7, 1998 as five days or forty hours. Measuring the distance between two points also results in a value represented by a different unit of measure that again may be an approximation. Consider the number posted on exit signs on the U.S. interstate highway system. This number represents an approximation of the number of miles from the origin point of the highway along the road's surface.

Arithmetic operation involving currency pose similar problems. Although currency may be added and subtracted to yield a value that is represented by the same unit of measure, division produces a result that does not. If a vending machine contains $.80 items and the patron has $2.00, the quantity of items purchased is 2, since the machine does not allow the purchase of half an item.

Solution. The basic strategy that can be applied to quantitizing values is to simply perform the arithmetic operation and then round the result according to a set of parameters. The first parameter specifies a method by applying the *Domain* rule pattern. For example, in most cases declaring a rounding method type to be the floor, ceiling, or nearest value. The other parameters specify the interval between values, the unit of measure, and the origin point. In many cases global default parameters may be specified and then overridden for the last two. For example, all time extents default to measurement in years and the origin point defaults to 0 for all extents.

When business rules dictate intermediate rounding rule because of a series of calculations, additional parameters may be required to fully specify the quantitizing process. For example, 1.4444444445 represents a number with ten digits of precision after the decimal point. Rounding this to the nearest integer seems straightforward, resulting in a value of 1. However, the result changes to a value of 2 if intermediate rounding rules progressively round the value to one less decimal point of precision.

Additional problems with intermediate rounding may result when operations are reversed. For example, dividing $ 1.00 by 3 results in a value of $ 0.33 rounded to two decimal points. Multiplying by three yields a $ 0.99 value rounded to two decimal points, which may not produce the desired result. Therefore, constraints on intermediate rounding may be required to avoid this situation. This represents dependency of one parameter on another. Specifically, an intermediate rounding must contain sufficient decimal precision to accommodate all multiplication operations within a valid domain range for the value. This can be accomplished by applying the *Domain* business rule pattern to establish a minimum and maximum value for the attribute and accommodating the decimal precision as a function of the differences in scale between the minimum value and the maximum value. However, this represents the worst case scenario. A more pragmatic approach is to only consider the minimum value for the attribute in its current state, but this represents additional parameterization that may be unnecessary.

Dates and calendars warrant special considerations. In additional to rounding rules that are applied to the values that between two points in time, the points in time themselves may need adjustment. The point may be adjusted to the beginning of the day, the first day of the month coincident with or in the following month, or the end of the year to address specific calculation needs.

Example – Employment Service. This example demonstrates how adjustment methods play a role in calculating the value for an interval. It also shows how the selection of one parameter carries with it the inclusion of another parameter and how a single parameter is used to abstract the usage and specification of a pair of complementary parameters.

The employment service calculation below assumes that an employee begins employment on the first work day of the calendar year and terminates on the last day of work in the same year. Company policy stipulates that an employee is entitled to one week of vacation after completing one year of service. Other benefits, such as insurance coverage eligibility, continue through the last day of the following calendar quarter.

This example involves defining a calendar that identifies the work days and non-work days, including holidays. The calendar may be logically represented as a list of dates and associated Boolean value that is set to true for workdays. This calendar is used in an adjustment that calculates service for each year. A pair of date-time adjustment methods called earliestNonworkingDayInMonth and lastestNonworkingDayInMonth is used. They are complementary methods specified by a single parameter. The algorithm for the first is shown as part of Example 6.3.

The calendar most likely will be established as the default calendar, so it would not be necessary to specify it for each application of the *Interval* pattern which required referencing it. Without the adjustment method and related calendar, the result would be 0.997 years.

Algorithm:

```
Function datetime earliestNonworkingDayInMonth( datetime aDatetime, calendar aCalendar ) {
    while ( ! aCalendar.isWorkDay( getCalendarDate( dateAdd( aDatetime, -1 ) ) )
        && dayOfMonth( aDatetime ) != 1  {
        aDatetime = dateAdd( aDatetime, -1 )
    }
    return beginningOfDay( aDatetime )
}
```

Parameters:	PeriodAdjustment:	IncludeNonworkingDayInMonth
	UnitOfMeasure:	Years
	Rounding:	Floor
	DecimalPrecision:	3
	Calendar:	

Date	IsWorkDay
1/1/97	False
1/2/97	True
…	
12/31/97	True

Case:	Input :	Start: 1/2/97	End: 12/31/97
	After Adjustment:	Start: 1/1/97 00:00:0000	End: 1/1/98 00:00:0000
	Output:	1.000 years	

Example 6.3. Employment Service Parameters and Case Values

The designation of (1/1/98 00:00:0000) as the end of the month of December has subtle implications regarding its treatment in other business rules, such as the insurance benefits. Points in time are dimensionless. Therefore, this representation does not imply any time worked in the subsequent year, which would change the cut-off of insurance coverage eligibility in this example from Mach 31 to June 30. The interpretation of (1/1/98 00:00:0000) depends on its context. In this context, it represents a point in time. In another context, it may represent shorthand notation for the 1/10000 of a second interval between (1/1/98 00:00:0000) and (1/1/98 00:00:0001).

Other period adjustment types would require their own collateral parameter. For example, a thirty-day month adjustment may apply, such that 2/7/97 to 3/6/97, 5/1/97 to 5/30/97, 1/21/97 to 2/19/97 and 2/1/97 to 2/28/97 all represent one month. This adjustment disregards the number of days in any month when the corresponding point in time in the following month is considered, but recognized as a full month from the first to the thirtieth for those months containing thirty-one days. Thus, although this example utilized an actual calendar basis, other basis types are possible and may be specified with a parameter if needed.

Example – Fuel Economy. A gas station pump displays gallons dispensed to decimal places of precision. Mileage is recorded between fill-ups based on a digital odometer that registers miles traveled to one decimal place. Each of these three values already represents a rounding method of type *floor*, however the purpose of this example is to illustrate the origin and interval parameters. With the inclusion of an intermediate rounding method for mileage, this calculation chains together two approximations.

Parameters:	UnitOfMeasure:	Miles	
	Rounding:	Nearest	
	Interval:	10	
	IntervalOrigin:	0	
	DecimalPrecision:	0	
Case:	Input : Start:	42134.3 miles	End: 42411.1 miles
	After Calculation:	276.8	
	Output:	280 miles	
Parameters:	UnitOfMeasure:	Miles per gallon	
	Rounding:	Floor	
	DecimalPrecision:	0	
Case:	Input:	280 miles / 12.24 gallons	
	After Calculation:	22.875	
	Output:	22 miles per gallon	

Example 6.4. Fuel Economy Parameters and Case Values

UML Representation and Related Patterns. The *Interval* business rule pattern is specialized based of the nature of the measurement. The two types described Example 6.3 and Example 6.4 are shown as the *NumericInterval* and *PeriodInterval* model elements.

This pattern shares similar concepts to the Weisert's Point-Extent Pattern [Weis97] and Fowler's *Schedule* pattern [Fowl97]. The *Point-Extent* Pattern is an implementation level pattern that addresses operator overloading. The *Schedule* pattern addresses the handing of recurring events for calendars. The *Domain* business rule pattern may be integrated to define the domains for each enumerated attributed and for defining minimum and maximum values for decimal precision to avoid loss of precision that may occur when chaining this pattern for determining intermediate values.

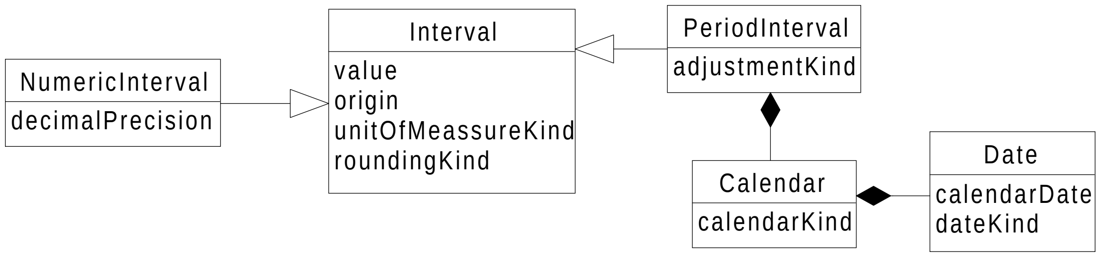


Figure 6.3. UML Representation of Attribute Interval Pattern

6.3.3. Rule Pattern: Domain

Problem – Describing Valid Values for an Attribute. The state on an entity can be determined at any given point in time by the values of its attributes. In certain states, attributes may be restricted in the values considered valid for that state. For example, a person sitting in a car on his driveway should expect the speedometer to read 0 miles per hour. However, there are exceptional conditions that may alter this observation. If a car is in an auto mechanic shop and the car is supported such that the wheels are free, this stationary car may register a value on the speedometer that doesn't reflect the current situation. Conversely, the person may be sitting is his car while being ferried on a boat. In this case, the vehicle is moving relative to the earth's surface but not showing on the speedometer. Finally, the car may be performing stunts off a ramp that casts it into the air is such a way that the speed and speedometer do not match. Although normal behavior may be well defined, other states represent valid values in exceptional states. If is often desirable that the exceptional states be considered valid, but also maintain constraints for normal behavior. Further difficulty arises when the value becomes the determining factor in establishing the exceptional state.

Solution. Describing valid states for an attribute can be addressed by establishing valid value domains for attributes of an entity. If the domain is discrete, an enumeration of domain values may suffice. This is mostly applicable for state and type attributes. A numeric domain may be discrete or continuous. The most generic application of the *Domain* business rule pattern is for continuous values. Discrete numeric values are a subset of the continuous range though application of the *Interval* business rule pattern by defining origin and interval parameters. Subsets of enumeration domains may be specified through application of the *Type Membership* business rule pattern.

Global default domains can be established for basic attribute types such as date-time, currency, temperature, and age. Minimum and maximum values may be defined with various ranges by applying the *Range set* business rule pattern. The global default range may be applied as the default for all basic data types. The range can establish invalid and exceptional value boundaries. Other ranges may be established that override at the entity type level, a specific attribute level, or even at a specific instance level. Additional ranges may be based on certain states for an entity that. In this context, the state of the object is checked against the range boundaries specified for a state attribute. This state attribute may be derived from a number of other attributes, including the one under consideration.

The *Domain* business rule pattern utilized a layered set of parameters that cover three dimensions as shown in Table 6.5. Layering is primarily concerned with the scope of the parameter. Although this specification pattern does not dictate a specific implementation strategy, inclusion of the context dimension may be facilitated by an approach that readily supports determination of states based on combination of attributes. In this manner, exceptional behavior handing is possible without ruling the value immediately as invalid. In other words, the response for a given scope may be to trigger exception handling, which in turn determines the state and checks to see if these values can be considered acceptable. Additional processing or alternative behavioral logic may then be applied to pass the value as acceptable.

Table 6.5. Domain Rule Pattern – Parameter Dimensions

Dimension	Examples
Scope	data type, attribute type, entity type, specific attribute of an entity, specific instance, specific instance attribute
Context	Instance state
Response	Invalid, exception handling

Example – Phone Number Type. A very basic application of the *Domain* pattern is to simply enumerate the possible values for a type. In this example a phone number may be classified as any of the values in its domain without restriction. Thus, this represents a global rule that applies any attribute that is declared as a phone number type.

Parameters:	Type:	Phone Number Type
	Domain:	residence, business, cell, fax, pager, modem, interactive voice response (IVR)

Example 6.5. Phone Number Type Parameters

Example – Cash Deposit. A bank deposit may provide for a global default of the largest amount of currency that can be expressed. However, for the purposes of a deposit, a more restrictive range is established. Specifically, a negative amount may not be valid and precision is limited to the smallest currency unit. Denomination units do not apply to computing interest, which is subject to rounding by applying the *Interval* pattern.

This example shows a deposit that triggers an exception handing situation. This exception may be due to federal regulations or bank policy. Nonetheless, conditional behavior is invoked. It is even possible to dynamically override the cash deposit valid range once conditional branching in invoked if the *Self Modify* business rule pattern is realized in the implementation environment. This would allow a cash deposit of over the valid limit of $1,000,000 by applying an override to this particular deposit instance. Alternatively, the customer making the deposit may be included in a set that is not subject to certain processing, such as the result of another workflow process that placed certain pertinent documents on file that satisfied processing requirements. In this case, the allowable limit is relaxed somewhat. Note that only the specification of an exception is made. This specification remains implementation independent.

Parameters:		
	Global default for currency valid range:	$ (999,999,999,999.9999) to $ 999,999,999,999.9999
	Deposit valid range:	$ 0.00 to $ 999,999,999,999.99
	Cash Deposit valid range:	$ 0.00 to $ 1,000,000.00
	Cash Deposit exception range:	$ 0.00 to $ 10,000.00
	Cash Deposit w/documents on file:	$ 0.00 to $ 50,000.00
Case:	Input:	Deposit of $54,376.63 of which $ 12,316.82 is by cash and the remainder by check
	Output:	Exception

Example 6.6. Cash Deposit Parameters and Case Values

UML Representation and Related Patterns. The *Domain* business rule pattern is a single model element represented though identification of its scope, context, and response attributes as shown in Table 6.5. No other structural connections exist in the basic pattern. Collaborations with other model elements may only exist through composition of the *Domain* pattern with other business rule patterns. The *Interval* pattern may be used in conjunction with discrete numeric values. The *Type Membership* pattern enables naming of subsets of enumeration domains specified by the *Domain* pattern. The *Self-Modify* pattern collaborates with the *Domain* pattern thresholds to dynamically modify behavior.

6.3.4. Rule Pattern: Type Membership

Problem – Describing Attribute Type Compatibility. An attribute is often classified as a particular type by industry conventions. However, this classification may carry additional semantic information that can more accurately be modeled as separate attribute facets. Rather than force unnatural conventions upon the users of a system that make it more difficult to use, this additional semantic information must be extracted from the classification. For example, in classifying an animal as a bird, an implicit assumption may be made that the subject of a classification can fly. However, penguins are birds and they can't fly. Bats are mammals and they can fly. Moreover, a pigeon may have a broken wing and in its current state, it cannot fly.

Solution. Determining if an attribute is compatible with a certain type can be accomplished through the declaration of a sub-type for modeling situations where only one or two attribute facets are relevant. For exceptional classifications, it may be necessary to attach a condition that checks the state of the entity, such as the broken wing that prevents a specific bird from flying. Conditions may be attached to entity types or instances. In the latter case, the set of name instances that possess the same attribute represent their own unique sub-type that can be chained with the simple enumeration sub-typing. This is especially useful when addressing modeling problems where grandfathered classifications exist, such as a group of active employees that were parties to a victorious class action suit against their employer which warrant special treatment. Such a group represents a closed set that may not consider additional instances.

Example – Passing Grade. A basic example that utilizes a sub-typing scheme creates a subset of all grades and then associate an attribute of passing with each. No additional conditions are needed to make the determination.

Parameters:	
Enumeration Type:	Grade
Enumeration Sub-type:	Passing grade
Enumeration Values:	A, B, C
Case: Is this a passing grade?	
Input:	Grade = B
Output:	True

Example 6.7. Passing Grade Parameters and Case Values

Example – Multi-Family Real Estate. Conditional logic is usually necessary to address exceptional situations. This example identifies a residential single family home and is able to include it as a member of multi-family because of qualifying conditions. Thus, this property exhibits classification in two normally disjunct categories. The classification of real estate property can become quite complex. The ability to identify an individual instance as an exception provides a mechanism to focus on the exceptions rather than a contrived classification that would be too onerous for normal usage. However, many exceptions of this type may be found in certain communities. Thus this example names an attribute that can identify a significant subset or real estate properties. If this was not the case, the individual property could be named as the value for the condition. This represents instance level parameterization.

Parameters:		
Enumeration Type:	Residential Real Estate Property	
	Domain:	{vacant, multi-family, attached, detached house}
	Enumeration Sub-type:	Multi-family property
	Value:	Detached house
	Condition:	has coach house
Case:	Is this a multi family property?	
Input:	123 Main Street – a colonial house with coach house over garage	
Output:	True	

Example 6.8. Multi-Family Real Estate Parameters and Case Values

UML Representation and Related Patterns. The *Type Membership* business rule pattern may be represented as a collection of types to which it is a member.

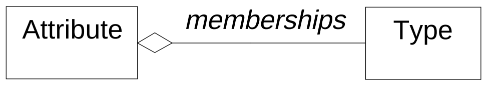


Figure 6.4. UML Representation of Type Membership Pattern

6.3.5. Rule Pattern: Default Value

Problem – Establishing an Initial Value if Unknown. If a value for an attribute is unknown, certain assumptions may often be made so that processing may proceed. In many situations, the presence of a value merely indicates a preferred method of handling a particular workflow. Therefore, processing should not be interrupted simply because this preference is not known.

Solution. Assigning a default value for an attribute provides a way for a system to proceed with processing of a workflow. Processing may elect to take a worst case scenario approach. In this case an additional workflow scenario may be activated to adjust the value when it is known. For some scenarios, the assignment may not be a constant. Instead, a method is used to determine the value that should be assigned.

Example – Mailing Address. An industry trade publication offers free subscriptions to professional that complete their application. Both an office and home address are provided, although only the office address is required. If both addresses are completed, a checkbox indicates which addressed should be used for mailing the publication. If neither is checked, the office address is used.

Parameters:		Attribute:	mailing address
		Type:	mailing address type {office, home}
		Default:	office
Case:			
	Input:	Attribute:	office address – known
		Attribute:	home address – known
		Attribute:	mailing address – unknown
	Output:	Attribute:	mailing address – office

Example 6.9. Mailing Address Parameters and Case Values

Example – Hotel Rooming Assignment. A conference is offering hotel accommodations as part of its package. An attendee may specify his rooming partner; otherwise one will be assigned. If there are an odd number of attendees, one will be selected to room alone. The remaining unmatched will be assigned based on certain similar attributes, such as employer, age, position, etc. In this example, the roommate preference is not made, so the assignRoomsMethod default behavior is invoked to make the assignment.

Parameters:		Attribute:	roommate
		Type:	registration Id
		Default:	assignRoomsMethod
Case:			
	Input:	Attribute:	roommate – unknown
	Output:	Attribute:	roommate – 251

Example 6.10. Hotel Rooming Assignment Parameters and Case Values

UML Representation and Related Patterns. The *Attribute Default Value* business rule pattern maps to the *defaultValue* attribute of the *Parameter* model element. This pattern provides specification for the *Uninterpreted* type with the *defaultValue* attribute resolved through the *body* attribute of the *Expression* model element.

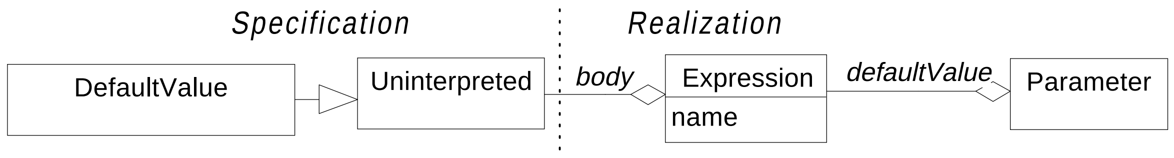


Figure 6.5. UML Representation of Default Value Pattern

6.3.6. Rule Pattern: Multiplicity

Problem – Describing Cardinality and Collection Count Restrictions. The state of an entity may require that a value be assigned to an attribute because it provides crucial processing information. In certain situations, such an attribute represents a collection of values in which not only is at least one value required, but additional constraint as to exactly how may values should be present must be stipulated.

Solution. Multiplicity is one of the model elements defined in the Data Types package of the UML specification. The UML Multiplicity model element is defined as a collection of one or more ranges of integers representing upper and lower bounds. This representation allows modeling of the most general case, but cannot by itself represent all situations. Thus a required attribute can be specified as having a single multiplicity range with an upper and lower bound of one. However, specification or a scalar value can be more succinctly expressed as either required or optional. If a global default for all attributes is set to optional, then only the designation of mandatory is necessary.

Enumeration of common multiplicity types can also be defined using the *Domain* business rule pattern. Such a domain should include optional (0..1), required scalar (1..1), any (0..n), required vector (1..n). By integrating the use of an *Interval* business rule pattern repeating counts such as only even, or only divisible by five can be expressed in concise notation.

<u>Example – Employee Name.</u> This example represents application of a mandatory scalar. A name is required for each employee. Although the name is comprised of several component parts, the presence of a name may be specified as required as a whole. Additional business rules may apply to these various parts that may also include reapplication of the multiplicity rule, such as first name and last name are required, but middle name, prefix (e.g. Dr.) and suffix (e.g. Jr.) are optional. In most cases, specification of the component parts is sufficient. This example specifies both for illustration purposes. Exceptional situations may occur where a person only has a first name or the name is an unpronounceable glyph and is captured only as an image. If these exceptional situations occur with sufficient frequency, then all fields may be optional and satisfactory combinations of component attributes may need to be specified using the *Association* business rule pattern.

Parameters:	Employee.Name:	required scalar
	Employee.Name.Last:	required scalar
	Employee.Name.First:	required scalar
Case:	Input:	Employee.First – Russ
		Employee.Last – unknown
		Employee.Suffix – Ph.D.
	Output:	Invalid name

Example 6.11. Employee Name Parameters and Case Values

<u>Example – Excursion Booking.</u> A high adventure excursion outfitter sponsors a rafting trip that can use one of two sized rafts. The first holds up to twelve people and the second up to thirty people. For

economic reasons, a minimum of six is required on the smaller craft and a minimum of eighteen is required on the larger one. This example demonstrates how one parameter can be used to specify another. The capacity is fixed for a particular type of raft, but the lower end may vary depending on other factors, such as the break even cost and excursion fee for each person.

Parameters:	Bookings:	[Breakeven bookings on small raft] – [small raft capacity]	
		[Breakeven bookings on large raft] – [large raft capacity]	
Parameter Binding Input:		Breakeven bookings on small raft:	6
		Breakeven bookings on large raft:	18
		Small raft capacity:	12
		Large raft capacity:	30
Parameter Binding Output:		Bookings:	6 – 12; 18 – 30

Example 6.12. Excursion Booking Parameters and Case Values

UML Representation and Related Patterns. The *Multiplicity* business rule patterns may collaborate with the *Domain* and *Interval* business rule patterns to extend the expressive capabilities of UML notation.

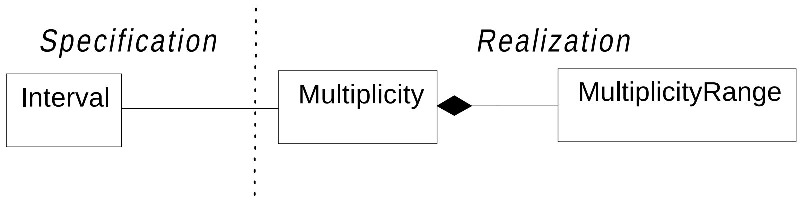


Figure 6.6. UML Representation of Multiplicity Pattern

6.3.7. Rule Pattern: Ordering

Problem – Establishing the Relative Sequence of Elements. The course of action taken in a business process is often determined by the order in which items are considered. For example, eligibility for employee benefits that are based on years of service must consider employment events that are ordered chronologically. This ordering permits a distinction between periods where service is credited and not credited. Several ordering sequences may also apply to the same collection of items. If a proposed new product relied on a customer opinion poll to assist in determining features to include, then each evaluated aspect could provide a ranking. Various weighted ranking that combined all of the aspects could also be generated. Therefore, correct handling requires that items conform to some ordering scheme that evaluates a relative property of one member to others.

Solution. The relative sequence of elements may be established through definition of a sort key and a functional restriction. A sort key requires the identification of an ordered set of attributes, a definition of an inequality function, and a direction. The set of attributes may be a combination of actual and derived attributes. The inequality function will typically be a binary comparison operation that established that one item is less than, or precedes, another item. Direction allows the ordering to be reversed, thus can be represented as a Boolean parameter, such as *isAscendingOrder*.

A functional restriction identifies a subset of the ordered elements when evaluating if a certain condition exists. The subset is identifies through an element count and direction, such as the preceding three elements or the adjacent two elements in both directions. The condition expression may be based on a binary evaluation that compares the current element against each of the other elements. Optionally, the *Meta-attribute* business rule pattern may be applied to a single comparison by considering the other elements through the aggregate attribute defined in that pattern.

Example – Emergency Contact. An organization identifies a person on call for handling reported problems with its database servers**.** Two additional backup persons are also required to be available on call. This example also includes application of a *Routing* business rule pattern by naming roles for the ordering rather than individuals.

Parameters:	sort key:	call order
	IsAscending:	<default:true>
Case:	CallOrder	Contact
	1	[database server primary contact person]
	2	[database server first backup person]
	3	[database server second backup person]

Example 6.13. Emergency Contact Parameters and Case Values

Example – Test Scores. A training program allows employees to proceed to the next instruction unit only if their last score is over 85 and they have shown improvement over two of the three preceding quizzes after completing the current unit. In this example, the last quiz fail to exceed two of the previous three scores, so an additional quiz must be taken, along with any other exercises mandated prior to taking the quiz. The next quiz will require a minimum score of 87.

Parameters:	sort key:	chronological date of quiz	
	IsAscending:	<default:true>	
	Functional restriction:		
		Scope:	3
		Direction:	preceding
		Count:	2
		Condition:	current.score > [element].score
Case:	Quiz date	Test scores	
	2/12/1998	85	
	2/13/1998	90	
	2/16/1998	86	
	2/17/1998	86	

Example 6.14. Test Scores Parameters and Case Values

UML Representation and Related Patterns. The *Ordering* model element contains an optional *FurnctionalRestriction* that constrains *Elements* that belong to a collection.

The *Ordering* business rule pattern may collaborate with the *Routing* business rule pattern by naming roles for the ordering rather than individuals.

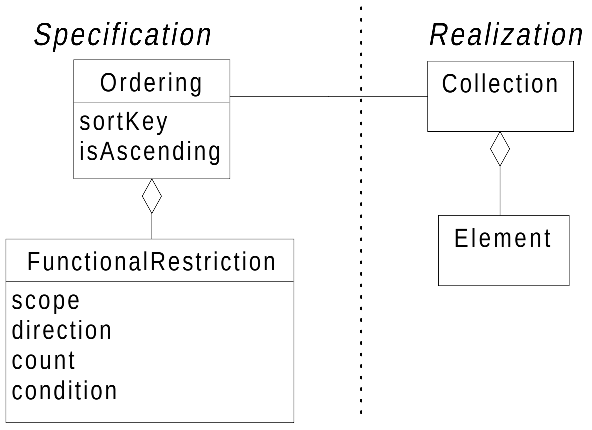


Figure 6.7. UML Representation of Ordering Pattern

6.3.8. Rule Pattern: Uniqueness

Problem – Describing Permissibility of Equivalent Elements. Business processes often rely on maintaining uniqueness over a collection of entities. Although an identification attribute may be assigned for internal use by a system, other attributes may still need to be unique across the collection that is independent of any implementation strategy. Uniqueness may apply to the whole collection of merely to a selected subset of the collection.

Solution. The first step in determining if elements are equivalent is to identify a set of one or more attributes of the elements of a collection that must be unique. Then, a Boolean equivalence comparison function must be defined. Finally, a scope for the uniqueness constraint must be defined as a function that selects a subset of the entire collection.

Example – Real Estate Listing. A multiple listing service issues a listing number for each property placed on the market by one of its member brokers. This number must be unique across the entire collection of all listings ever submitted. However, another unique restriction is placed on the property that requires that a property can only be actively listed with one broker at a time. A second attribute, the real estate tax number, is identified to enforce this restriction.

Parameters:	Attribute:	listing number
	Scope:	<default:all>
Parameters:	Attribute:	real estate tax number
	Scope:	is Active Listing

Example 6.15. Real Estate Listing Parameters

Example – Employee Identification Number. The headquarters for a corporate conglomerate is consolidating the benefits administration for all of its divisions. Since many of these divisions were previously independent companies, each has it own representation for an employee identification number. A combination of attributes will not suffice, such as division and employee number, since it is desired to maintain a unique number for each employee. Therefore, a new attribute must be created to uniquely identify each employee at the conglomerate level to overcome different employee numbers that result from a transferred employee from one division to another.

Parameters:	Attribute:	corporate employee number
	Scope:	<default:all>
Parameters:	Attribute:	employee number
	Scope:	Division

Example 6.16. Employee Identification Number Parameters

6.3.9. Rule Pattern: Recursion

Problem – Describing Recursive Ownership Traits. Organizations may place arbitrary restrictions on the depth of hierarchical structures or the number of links in a chain of relationships. One illustration of this nature would be an organization that desires to flatten its management structure by having at most three layers of management between any employee and the president. The desire is often to limit the complexity of business processes or to improve service.

Solution. Recursion limits may be put in place by defining a type and depth restriction for member elements.

Example – Thesis Sub-Headings. The thesis examiner requires that a graduate thesis may only contain four levels of sub-headings without approval from the student's advisor.

Parameters:	Type:	sub-heading level
	Depth:	4

Example 6.17. Thesis Sub-Headings Parameters

Example – Technical Support. An organization policy stipulates that a when a support request is reassigned more than twice the supervisor must be notified.

Parameters:	Type:	case worker
	Depth:	3

Example 6.18. Technical Support Parameters

UML Representation and Related Patterns. The *Recursion* business rule pattern provides constraints on the *CollectionElement*. The *depth* attribute may be further specified through collaboration with the *Domain* and *Multiplicity* business rule patterns.

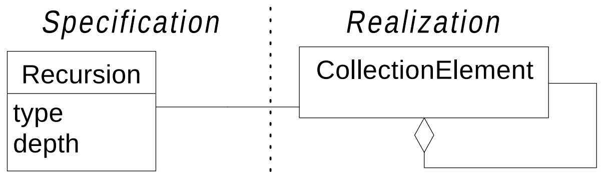


Figure 6.8. UML Representation of Recursion Pattern

6.3.10. Rule Pattern: Aggregation

Problem – Describing Aggregate Properties. Trends can be detected over a sufficiently large number of sample cases. Once identified, these trends may be used to take corrective actions. Performance requirements are often stated in terms of such aggregate properties.

Solution. A derived meta-attribute may be defined at the collection level that aggregates one or more attributes of the collection. The type of aggregate function and a threshold are defined in order to detect a condition in which action should be taken.

Example – Test Question. In order in guarantee that a test question is fair, each question is evaluated for overall average number of students that answered the question correctly. If this fall below the 50% threshold, the question is thrown out. Furthermore, this average is compared against a frequency distribution of how each of five percentile groups fared on the question. The question is also thrown out if any higher percentile group averaged less than 25% below any lower percentile group. This last part applies the *Association* and *Ordering* business rule patterns in order to define the threshold.

Parameters:	Aggregate function type:	average
	Threshold:	50% of answers correct

Example 6.19. Test Question Parameters

Example – Customer Phone Support. An organization's service departments requires that if the average delay for support calls exceeds fifteen minutes that corrective actions be taken, such as shifting telephone operators from another product line.

Parameters:	Aggregate function type:	average
	Threshold:	5 minutes

Example 6.20. Customer Phone Support Parameters

UML Representation and Related Patterns. The *Aggregation* business rule pattern is modeled in the UML as a derived attribute of a model element that contains a collection. The *Association* and *Ordering* business rule patterns may collaborate with this pattern to define the *threshold* parameter.

6.3.11. Rule Pattern: Sub-typing

Problem – Describing Type Membership Restrictions. Many classifications of an entity are really attribute facets that carry the potential for a combinatorial explosion of permutations. However, many of these permutations are not valid. Therefore, a need exists to more concisely represent the legitimate combinations of attribute values that allow all permissible one while disallowing those that are not valid.

Solution. Valid logical sub-typing is accomplished through the definition of disjunction and overlapping associations between sub-types. A set of sub-types is declared along with a disjunction or conjunction operator. If an entity can be expressed as possessing properties of more that one type of a given attribute, then overlapping may occur. If a mutual exclusion exists, a disjunction declaration is mandated. Mutual exclusion or overlaps may occur within subsets of the same descriptive facet of the entity. In these cases a single attribute type is declared and the logical connector applies across the entire set declared. In other cases a logical connector is applied across two different attribute types. Accordingly, both types are declared along with the affected domain values of their respective types.

Example – Employee Pension Benefit Types. An employee may have three types of pension benefits: death, disability, and retirement. However, the employee may receive payments for his disability or retirement benefits at any point in time, not both. The employee's beneficiary may receive death and retirement benefits concurrently, but can never receive disability benefits. Provided other conditions are met, such as the employee is declared disabled under the definition of the pension plan, the benefit types that the employee may receive is represented by a disjunction of these two benefit types. The beneficiary's payments represent a possible conjunction, or overlapping of benefit types.

Parameters:	Attribute Type:	Benefit
	Logical Connection Type:	Mutual Exclusion (OR)
	Type Set:	{disability, retirement}
	Condition:	is eligible, is disabled, is in payment…
Parameters:	Attribute Type:	Benefit
	Logical Connection Type:	Overlap (AND)
	Type Set:	{death, retirement}
	Condition:	is eligible, is deceased, is in payment…

Example 6.21. Employee Pension Benefit Type Parameters

Example – Condominium Lot Size. A condominium form of ownership is normally associated with an apartment, or attached housing in the real estate industry. This example reflects that observation. However, single family detached homes may also be owned in condominium. In such cases, a lot size may be appropriate to identify easement rights rather than outright ownership, similar to an assigned space in the parking lot.

Parameters:	Attribute Type:	Property
	Type Set:	{condominium}
	Condition:	not detached
	Logical Connection Type:	Mutual Exclusion (OR)
	Attribute Type:	Lot size
	Type Set:	{<ALL>}

Example 6.22. Condominium Lot Size Parameters

UML Representation and Related Patterns. The *Sub-typing* business rule pattern maps to the UML predefined *overlapping* and *disjoint* generalization stereotypes. This pattern may collaborate with the *Type Membership* business rule pattern to constrain type membership.

6.3.12. Rule Pattern: Association

Problem – Describing an Association Restriction between Entities. Business processes are often affected by the existence or state of certain entities in relation to other entities. Restrictions between entities may take one of two basic forms. The first form prevents the existence of one entity unless another entity does or does not exist. The second form constrains the state of one entity based on the state of another entity. Identification of such dependencies is necessary.

Solution. Define constraints between entities using the logical connectives: and, or, if exists, for all. The construction of parameters for this business rule follow the same structure as the *Sub-typing* business rule pattern between two attribute types. The only difference here is that the operators are applied between two separate entities rather than the same entity. The distinction may appear subtle, however separation into two similar business rule patterns enables one to distinguish between conceptual modeling entities.

Example – Custom Ad Layout. A franchisor provides a set of advertising templates that any dealer may use. Requests for customization of a template may be granted based on a fixed rate schedule of time and materials. Since only few requests are received each year, the franchisor does not employ an ad layout designer. Instead, when a request is received, an ad layout designer may be contracted to assist in the request. This restriction may also be expressed in this manner: an ad layout designer must not be contracted for services unless a dealer request exists.

Parameters:	Controlling Entity:	Ad Customization Request Order
	Dependent Entity:	Ad Layout Designer Contract
	Logical Connector Type:	Exists

Example 6.23. Custom Ad Layout Parameters

Example – Election Returns. The three candidates receiving the most votes win election to the board of directors. This example combines the *Ordering* and *Range* business rule pattern with a logical connector of 'for all'. Ordering in declared in descending order and a range of the top three elements is declared. Membership in that range determines the winning candidate. Only the parameters for the *Association* pattern are shown.

Parameters:	Controlling Entity:	Votes
	Dependent Entity:	Candidate
	Logical Connector Type:	For All

Example 6.24. Election Returns Parameters

UML Representation and Related Patterns. The *Association* business rule pattern is realized in the UML as a *Constraint* model element. The *Association* pattern is similar to the *Sub-typing* business rule pattern. *Ordering* and *Range* business rule patterns may collaborate to fully specify the association restriction.

6.3.13. Rule Pattern: Temporal Association

Problem – Describing a Temporal Restriction of One Entity to Other Entities. The situation often occurs that one entity must come into existence or reach a certain state before action can be taken on another entity. The existence or state of the first entity may prevent another entity from coming into existence. If the second entity is permitted to exist, its transition to a new state may be dependent upon the current state of a controlling entity.

Solution. Temporal restrictions can be specified through declaration of existence constraints. These existence constraints are expressed through temporal ranges and points. Temporal comparative operators, such as *before*, *after*, and *during*, are used to express timing requirements.

Example – Team Leader. When forming a new project team, an organization stipulates that the team leader must be designated before any other team members are named. This places an existence dependency on team members directed towards the team leader.

Parameters:	Controlling Entity:	Team Leader
	Dependent Entity:	Team Member
	Temporal Association:	Before

Example 6.25. Team Leader Parameters

Example – Price Reduction. A multiple listing organization prohibits price reductions on real estate property if the property is not actively listed. In this example, the dependent entity is allowed to exist, but is restricted from reaching the "processed" state if the state of the listing is not in an "active" status.

Parameters:	Entity:	Real Estate Listing
	State:	active
	Dependent Entity:	Price Reduction
	State:	processed
	Temporal Association:	During

Example 6.26. Price Reduction Parameters

6.3.14. Rule Pattern: Entity State History

Problem – Describing a Temporal Restriction of One Entity to Itself. Evaluation of a series of events is frequently required to determine the state of an entity in terms of acceptable values for attributes. The pattern of prior states or events may also dictate permissible courses of action.

Solution. An entity's temporal history can be defined through ordering rules that are expressed in terms of functional restriction, scope, and direction. Temporal ranges and points are used to establish conditional thresholds. Some applications of this pattern may require consideration of a single temporal range; other may require two or more.

Example – Medical Permission. A student who misses school for more than five days due to illness may not return without submitting a medical release from his doctor.

Parameters:	Attribute Type:	Attendance
	Prior Temporal Range status:	absent
	Threshold:	5 days
	Condition:	has medical approval

Example 6.27. Medical Permission Parameters

Example – Service Bridging. An employee that is rehired by a company may have his prior service reinstated provided he remains actively employed for a period that equals or exceeds the period of time between being terminated and being rehired. The parameters provided for this example assume a binary comparison of two temporal ranges. A more generic expression would name the temporal comparison operator in a manner similar to the *Temporal Association* patterns. By recognizing this type of comparison (i.e. between the current and prior ranges) as a common specialization of the more generic case, the number of parameters may be reduced.

Parameters:	Attribute Type:	Employment Status
	Prior Temporal Range status:	terminated
	Current Temporal Range status:	active
	Condition:	active >= terminated

Example 6.28. Service Bridging Parameters

UML Representation and Related Patterns. The *Entity State History* business rule pattern is realized in the UML as a *Constraint* model element. The *Constraint's* Boolean expression is constructed from the pattern's parameters.

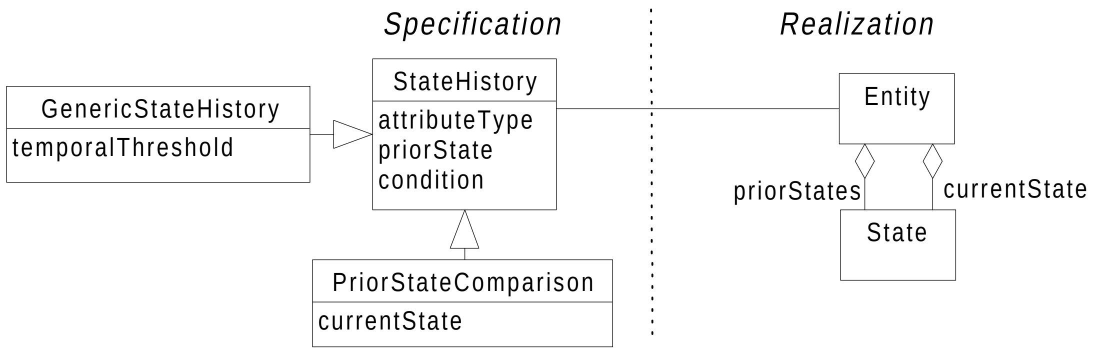


Figure 6.9. UML Representation of Entity State History Pattern

6.3.15. Rule Pattern: Temporal Aggregation

Problem – Describing a Temporal Aggregate Property of an Entity. The historical state or performance of an entity may reveal trends. Once identified, these trends may be used to take corrective actions. Performance requirements are often stated in terms of such aggregate properties.

Solution. Historical trends of an entity can be expressed though definitions of a temporal aggregate type (e.g. rates or counts). The history of the entity is considered a collection of prior states. A derived meta-attribute may then be defined at the collection level that aggregates one or more attributes of the collection over a specified period of time. The type of aggregate function and a threshold are defined in order to detect a condition in which action should be taken.

Example – Pension Estimate. A company policy restricts an employee to a maximum of four requests for pension estimates in a calendar year.

Parameters:	Aggregate Type:	sum
	Unit of measure:	pension estimate requests
	Period:	calendar year
	Threshold:	4

Example 6.29. Pension Estimate Parameters

Example – Sales Volume. A franchiser requires that a dealer must maintain an average monthly sales volume of over $20,000. If the average for any three-month period drops below $20,000, the dealer is placed on probation.

Parameters:	Aggregate Type:	average
	Unit of measure:	sales volume
	Threshold:	$20,000
	Period:	3 months

Example 6.30. Sales Volume Parameters

UML Representation and Related Patterns. The *Temporal Aggregation* business rule pattern is modeled in the UML as a derived attribute of a model element that contains a collection of its own states. It shares an affinity with the *Aggregation* business rule pattern in this respect.

6.3.16. Rule Pattern: Unit of Work

Problem – Describing a Set of Actions that must be performed in their Entirety or Not at All. Processes can be composed of many activities. Often some or all of these activities must be considered an atomic unit of work. In other words, if the set of activities cannot be completed, then the net effect should be as if they did not occur at all. The scope of these activities may be sufficiently fine grained that they can be considered in the same scope as a database transaction. In such instances, a simple rollback within the application instance may suffice. In other cases, the scope is sufficiently large that a compensating transaction or action must be executed to restore the prior state.

Solution. Define a collection of actions as an atomic transaction. A stable state is declared for the initiation of the atomic transaction that will be restored if the transaction is rolled back. The start activity and end activity are also identified to delimit the scope of changes.

Example – Posting Employee Data. A service bureau posts a client’s employee data in periodic batch jobs after receipt of a data file. Each file contains payroll, classification, and personal data records. If an employee that has records in more that one record format may end up with only partial posting if a record contains data that does not pass validation. The service bureau is required to either process all of the records for a given employee or none at all until data corrections have been made. This example demonstrates how a portion of a workflow process is rolled back while preserving the remainder.

Parameters:	Stable state:	employee data posted
	Start activity:	process personal data
	End activity:	process payroll data

Example 6.31. Posting Employee Data Parameters

Example – Pension Benefit Election. A company policy stipulates that a pension benefit election packet is prepared for each employee at time of termination if they are eligible to immediately commence receiving benefit payments. The employee has 90 days to make an election. If the election is not received within that period, the transaction is cancelled, leaving only the calculation results. This example rolls back the entire processing from the granularity of a single entity rather than the entire set of entities, or employees. Conditions may also be established that required the rollback of the entire batch.

Parameters:	Stable state:	calculation completed
	Start activity:	produce packet
	End activity:	process election

Example 6.32. Pension Benefit Election Parameters

UML Representation and Related Patterns. With the integration of adaptive use case extension to the UML, a single *UseCaseSegment* may be identified as a parameter to establish the scope of activities defined in the *Unit Of Work* business rule pattern.

6.3.17. Rule Pattern: State Transitions

Problem – Describing a Restriction on the Transition of One State to Another for an Entity. Business processes rely on certain conditions being fulfilled before an entity may move from one state to another state. Those conditions frequently include specified sequences of states that must progress in a prescribed order.

Solution. Valid state transition may be defined by means of ordered transitions that are characterized by transition types such as progressive, reinitiating, and cycle. Representation may take several forms, including a state transition matrix or state machine model. The manner used for parameter driven business rules is a regular expression grammar. This key aspect of this business rule type is the specification of the primary transition paths in order to describe the overall nature of the state transitions rather than an exhaustive definition of all permutations. Accordingly, it may be used as a starting point in defining a complete state transition matrix. This business rule pattern captures the distinctive elements of an entity’s life cycle in a manner compatible with the product of the event-sate cluster analysis described in chapter seven.

Example – Employment Status. The first status for an employee must always be active. Any other status may follow. A terminated status is a terminal status, but unlike deceased it can lead to a cycle of active status through a rehire event. Intervening status types include laid off, leave of absence, and suspended. Any of these must reinitialize to an active status or conclude a cycle with a terminated status.

Parameters:	Transition type: reinitializing
	Active.(leave of absence | suspended | laid off)
	Transition type: cycle
	Active.(leave of absence | suspended | laid off).terminated
	Transition type: progressive
	Active.(retired | deceased)

Example 6.33. Employment Status Parameters

UML Representation and Related Patterns. The *State Transitions* business rule pattern maps to the *Transition* model element.

6.3.18. Rule Pattern: Trigger

Problem – Describing "When Needed" and "When Changed" Actions. Many business processes rely on actions that must take occur when a value from an entity is needed or the state of the entity changes.

Solution. Triggers may be defined in the form of business rules by declaring watched attributes that require actions to be taken when accessed or when updated. This rule pattern is declared through a trigger type parameter along with a watched attribute. Define a when needed trigger and/or a when changed trigger

Example – Benefit Election Change. Once an employee's benefit commences payment, no further election changes may be made and all other calculations that had a status of pending election are invalidated. In this example a state change triggers an update to the state of other entities. Key values that trigger the action may also be specified if needed. This rule pattern may be used in conjunction with the *State Transition* rule pattern to specify the bulk of a state machine in terms of transitions and actions.

Parameters:	Trigger type:	when changed
	Watched attribute:	Benefit status
	Value:	in payment
	Action:	invalidate other calculations

Example 6.34. Benefit Election Change Parameters

Example – Accrued Benefit Request. When an employee inquires as to the amount of his accrued benefit and the value is not current, then a calculation based on the employee's employment data is run. This type of rule illustrates a when needed type of trigger.

Parameters:	Trigger type:	when needed
	Watched attribute:	accrued benefit amount
	Value:	<unknown>
	Action:	run benefit calculation

Example 6.35. Accrued Benefit Request Parameters

UML Representation and Related Patterns. The *Trigger* business rule pattern maps to the *trigger* association between an *Event* and a *Transition* model element.

6.3.19. Rule Pattern: Transaction Event Management

Problem – Describing Mutual Dependencies and Synchronization between Entities. An action may be blocked from being performed until two or more events occur. These may be simultaneous events or events that place the entity or entities in a satisfactory state.

Solution. Event management may be defined in terms of a stimulus-response or event-condition-action structure to capture initiating or prevention action logic. Since the *Event Management* pattern is defined as a mutual dependency, the participation of two or more entities is often present. Since events are the focus, temporal aspects are also present. The distinction of mutual dependency and synchronization types refers to the existence of events occurring in any sequence versus the simultaneous occurrence of events.

Example – Initiating Benefit Payment. The administrator for a pension trust fund is not authorized to commence payments until an election has been made and confirmation is received that the employee has terminated employment.

Parameters:	Event Management Type:	mutual dependency
	Events:	termination confirmation, benefit election

Example 6.36. Initiating Benefit Payment Parameters

UML Representation and Related Patterns. The *Event Management* pattern has an affinity with both the *Temporal Association* and *Entity Association* patterns. It differs from both of these primarily in the focus of the rule specification, which is event based rather than state based.

6.3.20. Rule Pattern: Deferred Action

Problem – Defining Actions prevented from being executed Immediately. Many times an event will occur that cannot be immediately acted upon. These events need to be tracked and handled at some later point in time.

Solution. Deferred actions may be specified through the definition of blocking and queuing types. A blocking type prevents any actions in response to the event until some condition is satisfied. A queuing type provides a mechanism for responding to a backlog of events.

Example – Web Based Customer Support. The customer service department for an organization manages a web site that enables customers to submit support requests completing a web page form. All requests are handled in a first-in, first-out manner. However, two separate queues are maintained, one for preferred customers and another for all other customers. Since preferred customers have paid for support, they are handled in a ratio of four to one. This example also applies the *Ordering* rule pattern to establish the four to once ratio of request handling between the two customer types. This is accomplished through a functional restriction, however the parameters are omitted for clarity of the *Deferred Action* rule pattern.

Parameters:	Preferred customer Queue type:	FIFO
	General customer Queue type:	FIFO

Example 6.37. Web Based Customer Support Parameters

UML Representation and Related Patterns. The *Deferred Action* business rule pattern maps to an *ActionSequence* that is associated with a *deferredEvent* between a *State* and an *Event*. It specifies the effect of the *Transition* that is triggered by the *Event*.

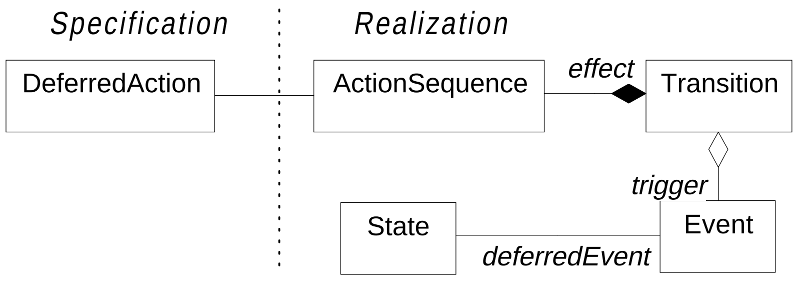


Figure 6.10. UML Representation of Deferred Action Pattern

6.3.21. Rule Pattern: Priority

Problem – Resolving Conflicts among Competing Actions that can occur as a Result of an Event. When a business process has several activities that can be performed simultaneously, some mechanism for indicating which activities are more important is needed. The priority of activities should be able to be specified in absolute terms or in relative terms to each other. The net result should indicate an ordering for performing the activities.

Solution. Define absolute and relative priority levels to handle conflict resolution.

Example – Listing Processing. A real estate company prescribes that upon securing a listing that a number of activities should take place. First, the listing should be submitted to the multiple listing service. A follow-up seller's packet should be sent before advertising copy and photos are produced. The priority of actions is dictate by legal obligations as well as good business practices.

Parameters:	Action:	submit listing to MLS	
		Priority type:	absolute
	Priority level:	1	
	Action:	seller packet	
		Priority type:	relative
		Relative placement:	before
		Relative action:	ad copy

Example 6.38. Listing Processing Parameters

UML Representation and Related Patterns. The *Priority* business rule pattern does not directly map to any UML model element. However, they are most closely related to *PseudoState* model elements that are designated as a *fork* kind. This pattern shares an affinity with the *Ordering* business rule pattern, however the focus here is on activities rather than members of a collection.

6.3.22. Rule Pattern: Scheduled Execution

Problem – Describing Actions that must occur at a Specific or Relative Point in Time. Maintaining schedules of required actions is a common business problem. Actions may be performed at a specific point in time, but often that point in time is influenced by the day of week or intervening holidays. Alternatively, the timing of performing an action may be relative to some event.

Solution. Scheduled execution of activities may be accomplished through definition of calendar types and duration subtypes. The calendar provides for absolute points in time. Scheduled execution of an action may then be determined from the base point in time and the extent, or duration. This concept of date-time manipulation was previously introduced in the *Attribute Interval* pattern.

Example – Case Resolution. A customer service department policy stipulates that any case that remains open for more than 14 days must be escalated in its priority status. In this example a standard 365 day calendar is used. This implies that no allowances are made for weekends or holidays. The policy may be restated as ten days with use of the corporate work calendar.

Parameters:	Schedule Type:	Relative
	Recurrence:	<No>
	Start Event:	Case Opened
	Duration:	14 days
	Calendar Basis:	365 Day

Example 6.39. Case Resolution Parameters

Example – Course Calendar. The training schedule for a calendar year must be prepared and distributed by the second Monday in November of each year. The use of dates established relative to a calendar is a common situation that relies on collaboration with the *Ordering* business rule pattern. In contrast to previous examples, the parameters for both are presented in order to illustrate the specialized application of the functional restriction parameter of the *Ordering* pattern. Essentially, the application of the functional restriction in the more generalized parameters can be translated as follows: Give me the member date in the month of November dates collection that is ordered by date and has exactly one member element preceding it for which the day of week is Monday. The shorthand parameters consolidate the condition of the member element with the preceding elements and set a more intuitive order parameter rather than preceding count parameter. This one parameter replaces five from the more general case.

Scheduled Execution Parameters:		
	Schedule Type:	Calendar
	Recurrence:	Annual
	Start Event:	Nov 1
	Order:	2
	Condition:	dayOfWeek = Monday
Functional restriction Parameters:		
	scope:	ALL
	direction:	preceding
	count:	1
	comparative:	=
	condition:	dayOfWeek = Monday

Example 6.40. Course Calendar Parameters

6.3.23. Rule Pattern: Role

Problem – Associating a Real World Entity with a Role. Employees may be assigned a variety of responsibilities within their organization. These responsibilities may be enumerated in a job description for the position they hold. Frequently, informal responsibilities are added that extend beyond the base job description. Other responsibilities may be dropped. Regardless of formal responsibilities, tasks are assigned and work is completed by individuals that would have never been involved if strict adherence to job descriptions was maintained. Delegation of activities must be flexible enough to recognize these informal responsibilities.

Solution. Actor role types and organization structure mappings may be created to associate employees or automated support tools with a role. Integrating the adaptive use case user view model with this business rule template provides a concise and powerful approach to completing such mappings. The use cases are specifically constructed to define the system boundary to include all elements that participate in a workflow except for a single actor. Job descriptions may be defined as a set of use cases and constructed for each employee. Through application of the *Priority* business rule pattern, a ranking of importance may be established.

The assignment of use cases to individual employees only provides a general basis for the actual binding of a use case instance to that individual. Since many employees may play the same role, some criteria must be used to make this determination. The *Ordering* pattern (Example 6.13) described how the binding of an actual instance is approached from the role side. Through the application of the *Collection Aggregate* and *Priority* patterns, work load balancing and timely handling of activities may be managed.

Example – Database Administrator. The position of a database administrator consists of a set of roles defined the collection of use cases. A company employs three people that share this same title. Since all three have identical sets of roles, *Ordering* and *Scheduled Execution* business rule patterns are also applied to rotate which one is the primary person on call at any given time. Thus, each is assigned a reference to the ordered collection that contains all employees that share their job description.

```
Parameters:     Employee Job Description Template:
                          Collection: <User View Adaptive Use Case>
```

Example 6.41. Database Administrator Parameters

UML Representation and Related Patterns. The *Role* business rule pattern maps to *Actor* model elements that are related to the specified *AdaptiveUseCase* in the adaptive use case UML extension. The *Role* pattern may collaborate with the *Priority, Collection Aggregate, Ordering,* and *Scheduled Execution* business rule patterns to establish bindings to use case instances.

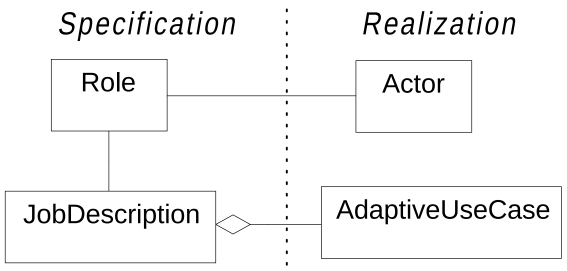


Figure 6.11. UML Representation of Role Pattern

6.3.24. Rule Pattern: Authorization

Problem – Describing Authorization and Action Scope. Authorization for an action often reflects formal organization structures. Levels of management control and chain of command influence how authorizations are granted. Permissions are frequently granted at a sufficiently fine-grained level to make distinctions between access, creation, modification, and deletion actions.

Solution. Action authorization may be defined through authorization types and actor sub-type mappings. Default authorizations are established that can be relaxed for roles higher up in the authorization hierarchy. Parameters define the scope in terms of create, read, update, and delete operations.

Example – Data Correction. The personnel department for an organization is responsible for maintaining data on all employees. Data is processed by them to keep employee classification, payroll, and personal data current. Historical corrections to this data are necessary from time to time. Any staff member making corrections must have their edits peer reviewed by another staff member. A supervisor making a correction does nor required review. This example also incorporates the *Range* rule pattern to distinguish between historical data and current data. The *Ordering* rule pattern places the supervisor at the top of the authorization hierarchy, which only consists of two levels here.

Parameters:	Role:	Staff	
		Modification:	all
		New Data:	no approval
		Historical Data:	peer review
	Role:	Supervisor	
		Historical Data:	no approval
	Hierarchy:		{supervisor, staff}

Example 6.42. Data Correction Parameters

UML Representation and Related Patterns. The *Authorization* business rule pattern maps an *Actor* model element. This pattern may collaborate with the *Range* and *Ordering* business rule patterns to establish authorization hierarchies and scope of authority.

6.3.25. Rule Pattern: Contingency

Problem – Describing Timed Window for Actions. A business process may prescribe that a certain action should be performed within a specified period of time. If that action fails to be performed within the stipulated period, for whatever reason, then a contingency action should be taken instead.

Solution. Contingency actions may be declared in association with a fixed period of time that defines the primary action window. This business rule pattern shares an affinity with the *Scheduled*

Execution pattern. The primary difference is that the scheduled action is contingent upon failure of a primary action occurring. However, the parameters from *Scheduled Execution* are subsumed by this pattern.

Example – Cash-Out Election. An employee benefits administration department processes cash outs for terminated employees that have an accrued value below a certain threshold. The terminated employee is allowed 90 days to make an election to roll over the cash value amount. If that election is declined or no election is made within the allowed period of time, then the cash balance amount will be sent after the mandatory government tax withholding has been deducted.

Parameters:	Primary Action:	cash-out election
	Contingent action:	cash-out

Example 6.43. Cash-Out Election Parameters

6.3.26. Rule Pattern: Enforcement

Problem – Describing the Strength of Rule Enforcement on an Action. Changing conditions often affect the permissibility of a process or action. When such volatility is present, a mechanism to monitor the changing condition is needed in order to enable or disable an action from occurring. Under certain circumstances, the process may need to be audited to make this determination.

Solution. Enforcement strengths can be established through the definition of enforcement types. Specifically, enforcement may take two discrete responses between universal permission and universal prevention of a given action. These responses are notification and logging. Notification is an active modification to the rule's enforcement that allows other actions to monitor or mitigate the impact of the primary action impact. Logging is a passive modification that merely audits the action. In this sense, this action applies the *Explanation* pattern.

Example – Purchase Order. Purchase orders that are submitted by non-management staff are subject to varying level of enforcement. If the figure is small, no approval is necessary. As the value get greater, more attention is paid. This example applies the *Range* pattern to establish threshold for various levels of enforcement. The key parameters are the enforcement type and the linkage to the respective range, which is governed by its own set of parameters. A further level of indirection through use of the

Ordering pattern should define an ascending set of four ranges so that the amounts could change independently of this rule.

Parameters:	PO $ Range	Enforcement
	$0	permit
	$10,000	log
	$50,000	notify
	$100,000	prevent

Example 6.44. Purchase Order Parameters

6.3.27. Rule Pattern: Self-Modification

Problem – Describing Reflective, Self-Modifying Behavior. Business rules may be defined with varying binding strengths. Contracts constitute an agreement enforceable by law. Directives are orders or authoritative instructions with specific authority. Practices represent a principal or accepted rule governing the procedures followed by an organization. Plans indicate a scheme or method of action for obtaining a specific goal. Opinions signify a belief of judgement that rests on grounds insufficient to produce complete certainty [Nass91]. The higher the binding strength, the less likely that the rule would be overridden or ignored. Additionally, the authority for the rule's creation influences its enforcement.

Solution. Binding strengths and proponents may be used to set conditional thresholds that determine under what circumstances a rule may be turned on or off. A rule conforming to the *Self Modification* business rule pattern can be considered a meta-rule that controls the nature of other rules. Although these declarative rules are implementation independent, the scope magnitude for certain rules may have such a significant impact on the environment that reconfiguration or recompilation of components may become necessary. In such cases a mapping of event to components can be defined.

Binding strength can play a major factor in the introduction of new rules. For example, the imposition of a federal regulation may nullify a departmental practice. The transfer of proponents from one functional area to another may place implicit reconfigurations on a business process that reflects their authority level and personal bias for workflow.

Example – Quality Control. A claims processing application is monitored by a quality control team. Each new adjustment calculation is audited. If each of the last fifty items passed inspection, then the inspection process in modified to only inspect ten percent of the items until another error is detected. The key parameters for this rule pattern are identification of a condition and target configuration component.

Parameters:	Condition:	last 50 pass inspection
	Target Configuration Component:	Random 10% parameter set

Example 6.45. Quality Control Parameters

6.3.28. Rule Pattern: Derivation

Problem – Describing a Mathematical Calculation. Almost all business processes involve some form of mathematical computation. When the calculation is subject to some degree of variability, the aspects that are subject to change need to be identified. Each piece of the algorithm must be bound to a concrete value in order to compute the result within the context of an individual calculation.

Solution. Mathematical calculations that exhibit any degree of variability may be declared as a parameterized algorithm type. Complex algorithms may require their own domain specific language.

Example – Highest X of Y. A company awards sales bonuses based upon the compensation of each eligible employee. Since the actual calculation to determine the base compensation varies to some degree, a generalized algorithm is used that is parameter driven. Typically the highest year out of the last five is used, but for some types of seasonal salespersons this is changed to the highest twelve consecutive months out of the last three years or the highest eight months over the last two years.

Parameters:	Included Periods:	1
	Averaging Period:	5
	Period Type:	calendar year
	Is Consecutive:	True
Parameters:	Included Periods:	12
	Averaging Period	36
	Period Type:	calendar month
	Is Consecutive:	True
Parameters:	Included Periods:	8
	Averaging Period	24
	Period Type:	calendar month
	Is Consecutive:	False

Example 6.46. Highest X of Y Parameters

UML Representation and Related Patterns. The *Derivation* business rule pattern maps to an *Operation* model element.

6.3.29. Rule Pattern: Inferred Fact

Problem – Describing an Inferred Fact. The deduction of facts from other facts is a basic feature of rule processing. When no calculation is involved in deriving the new fact, then a declaration must be made that one fact implies the existence of another fact. The configuration and classification of entities often rely on the ability to infer one fact from another.

Solution. Inferred facts may be declared though definition of a formal grammar structure. The individual rules production rules declared with grammar may be organized into rule sets. The base facts become typed antecedents and consequents used in induction and deduction operations. The declaration of these rules makes no assumptions with respect to how they will be used. However this business rule pattern parameterizes this aspect as well. For example, addressing configuration problems is typical use for forward chaining inferencing. Classification problems are best suited for backward chaining inferencing. However, the intent of this section is to describe the nature of this common business rule pattern and not to digress into the realm of expert systems and rule inferencing algorithms.

Example – Property Appraisal. A real estate company utilizes a search engine for identifying properties based on its close proximity of features to a subject property, which in this case is a two story single family residence. Based on key distinguishing attributes of the property, the applicable rule set and search strategy are executed.

Parameters:	Rule Set:	Residential Property
	Search Strategy:	Backward Chain

Example 6.47. Property Appraisal Parameters

6.3.30. Rule Pattern: Type Conversion

Problem – Describing Semantically Meaningful Type Conversions. Many times one entity must be able to stand in for another entity. In such instances a type conversion is necessary that makes the stand-in appear to be of the same as the desired type.

Solution. Type conversions may be accomplished through declaration of type hierarchy mappings. The assumption of this business rule pattern is that some common shared ancestry between two types exists. Thus, all common attributes are compatible by type membership. Each of the remaining differences is decomposed until only primitive attribute types remain. A mapping between related attributes may be either one-to-one or one-to-many. Conversion algorithms are introduced if necessary by applying the *Derivation* rule pattern.

In most cases a loss of information occurs, but the outcome may be sufficient for the intended purposes. For example, an attribute declared as a float loses precision if converted to an integer, but either representation may be sufficient for intended use in a calculation that would have rounded the number anyway. Therefore some measure of the tolerance of fidelity of the information can be indicated as part of the type conversion rule. If one considers image file formats, some formats such as PICT and TIFF are lossless representations of the original uncompressed bitmap. However, other compression formats make approximations for how the image is constructed, such as in the AVI and MPEG motion picture formats.

Example – Dual Use Real Estate. A residential property listed by a broker has a commercial zoning designation. In order to increase exposure and potential value, the broker markets the property as both a residential property and a commercial property. The source and target of the conversion and one of the attribute mappings is provided. The conversion expression may be expressed as a *Derivation* rule pattern and reused in other contexts.

Parameters:	Conversion Mapping Source:	Residential
	Conversion Mapping Target:	Commercial
	Mapping:	
	Commercial.squareFootage = sum(room.length x room.width)	

Example 6.48. Dual Use Real Estate Parameters

UML Representation and Related Patterns. The *Type Conversion* business rule pattern maps a *Type* model element. This pattern is closely related to the *Type Migration* business rule pattern.

6.3.31. Rule Pattern: Type Migration

Problem – Managing Evolution of Type Definition. Often business requirements change and the nature of attribute values for an entity are modified as a result. In some cases a new status or type is introduced. In others a change from mandatory to optional may be appropriate. When new attributes are introduced, some way must be found to fabricate the values for existing entities.

Solution. Type migration is addressed though definition of attribute mapping between type versions. This business rule pattern is identical to the *Type Conversion* pattern except that the same entity instance is involved in a *Type Migration* rule as opposed the creation of a new instance for a *Type Conversion* rule. The parameters identify each attribute that is changed along with the nature of the change (i.e. create, modify, or delete).

Example – Union Affiliation. The personnel department for an organization maintains employee records. Union affiliation is tracked as an indicator and used for inclusion or exclusion from certain benefits programs. Recent organizational changes introduced several new unions into the organization and now eligibility for benefits programs must be based on which union an employee is affiliated with instead on just an indicator of union membership. Accordingly the attribute type is change from a Boolean to a code. All existing employees are assigned the default code of the original union. The domain for the new attribute type may be declared in a *Domain* business rule pattern.

Parameters:	Attribute:	Union indicator
	Operation:	Modify
	Source Type:	Boolean
	Target Type:	Code
	Mapping:	
		IF True
		THEN union code = <CurrentUnion>
		ELSE union code = <None>

Example 6.49. Union Affiliation Parameters

UML Representation and Related Patterns. The *Type Migration* business rule pattern maps a *Type* model element. This pattern is closely related to the *Type Conversion* business rule pattern. It also collaborates with the *Domain* business rule pattern to define new attribute types domains.

6.3.32. Rule Pattern: Interpolation

Problem – Deriving a Point Value from a Range given Another Point Value and Range. Some business processes rely on calculations that must estimate a value that falls between two other values that provide an index into this desired value. This is often the case in lookup tables that are legally mandated or may have arisen out of traditional industry practice.

Solution. An interpolated value can be derived through enumeration of interpolation algorithm types. This pattern shares similarities with the *Interval* pattern, except here a continuous value is desired from provided discrete value rather than vice versa. Both patterns have floor and ceiling types. However, they are included only in this pattern for completeness. The same net effect is obtained from applying the *Interval* pattern first and then performing a direct lookup. The primary difference in the type domains is reflected in the opposite orientation of the two patterns. Instead of nearest, this *Interpolation* pattern defines a linear type.

Example – Tax Proration. A real estate contract stipulates that property taxes are to be prorated up to and including the day of closing based on the most recent tax assessment. A 365 day year is used.

Parameters:	Target Attribute:	Real Estate Tax Proration
	Dimensions:	1
	Source Attribute:	Day of Year
	Interpolation type:	linear

Example 6.50. Tax Proration Parameters

Example – Early Retirement Factor. A pension benefit is calculated and then reduced for early commencement before the normal retirement date for an employee. Based on the form of payment, if the spouse is designed as a surviving beneficiary the reduction amount takes into consideration both the ages of the employee and the spouse. A table provides lookup values in years. However, the plan calls for interpolation based on age in completed months. This example further highlights the similarities between the *Interval* and *Interpolation* patterns, since the age must first be quantitized using a floor interval rule before performing the linear interpolation on the table lookup.

Parameters:	Target Attribute:	Early Reduction Factor
	Dimensions:	2
	Source Attribute 1:	Employee Age
	Source Attribute 2:	Spouse Age
	Interpolation type:	linear

Example 6.51. Early Retirement Factor Parameters

UML Representation and Related Patterns. The *Interpolation* business rule pattern maps to the *Operation* model element. This pattern is closely related to the *Interval and Derivation* business rule patterns.

6.3.33. Rule Pattern: Protocol

Problem – Describing the Formal Syntax for Cooperating Components. Business processes rely of automated support that requires collaboration among components. In order for the components to properly communicate, formal syntax in terms of information exchanged must be established.

Solution. Protocol is established through definition of interfaces that prescribe an ordered sequence of arguments, types, and defaults. Specifications for each message exchanged, including return status and exception conditions are provided. Each argument is defined in terms of a parameter which provides the default value, the direction (in, out, or both ways), and the classifier type. The format chosen should conform to industry standards in order to provide the greatest degree of flexibility. The *Protocol* business rule pattern recognizes the importance of interface specifications. Such specifications are a major area of interests for consortiums of vendors. The nature of protocols is well understood and is included here for completeness. The examples provided here name two important interface specifications. Specific examples conforming to these interfaces can be found in those documents.

Example – Interface Definition Language. The Object Management Group's Common Object Request Broker Architecture Version 2.1 specification provides for a complete interface description through its Interface Definition Language (IDL). A formal grammar is used for specifications that consists of one ore more type definitions, constant definitions, exception definitions, and module definitions.

Example – Workflow Application Programming Interface. The Workflow Management Coalition Workflow Client Application (Interface 2) specification version 1.2 provides a mechanism for applications to access services of workflow engines using a consistent interface. A complete set of data types, attributes, and return codes are defined. Mappings to IDL bindings are also included.

UML Representation and Related Patterns. The *Protocol* business rule pattern maps to the *Argument* model element associated with an *Action.* The *Protocol* specifies the ordered list of *Arguments* attached to an *Action.* The *Action* in turn dispatches a *Request* that provides the specification for a *MessageInstance.*

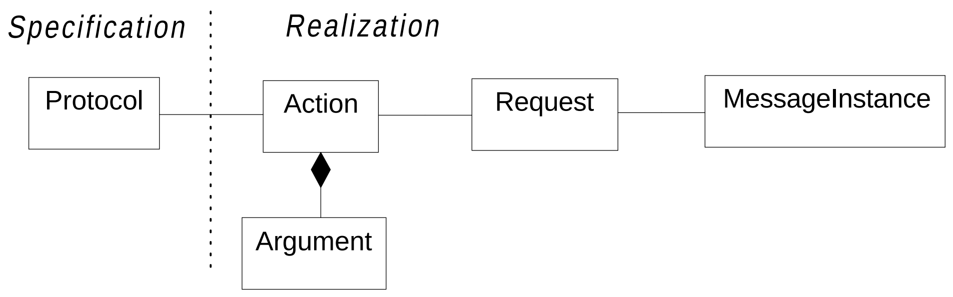


Figure 6.12. UML Representation of Protocol Pattern

6.3.34. Rule Pattern: Alternatives

Problem – Describing Alternative Actions. Business domain experts recognize the value of describing a business process that completely describes a typical sequence of activities. Alternative actions or extensions to this main typical course of action are often described separately so that the essence of the process description remain uncluttered with details that only become necessary when greater depth is needed to understand finer points of the process.

Solution. Alternative actions and extensions may be defined in terms of conditions and variant action sequences. Adaptive use cases are specifically modeled to address these needs. An explanation of how use cases may be parameterized and specific examples are provided in Chapter five.

6.3.35. Rule Pattern: Exceptions

Problem – Describing Exception Actions. When breakdowns occur in a business process, recovery actions are necessary to address exception conditions. Various levels of recovery are possible. Depending of the anticipated frequency of encountering such exceptions, appropriate effort should be expended to sufficiently address them.

Solution. Breakdown in normal processing can be addressed though defined types for exception handling intervention. Failures due to unanticipated conditions can be audited. When the business process integrates a mixture of human collaboration with automated systems, states can be defined to trigger manual intervention and override activities. When automated systems collaborate with each other, audit logs may be used to monitor breakdowns and take corrective action as needed. An enumerated set of exception flags may be defined using the *Domain* rule pattern.

Example – Manual Intervention Flag. An automated system calculates market value for a property based on current properties for sale, previous sold properties, and properties that were withdrawn from the market without selling. Certain types of property required intervention at certain points in the calculation process in order to compensate for incomplete information or features that fall outside the capabilities built into the system. In such cases manual intervention flags are set to allow human intervention to review the calculation in progress. Values may be provided, accepted, and overridden as needed. The following is a sample set of manual intervention flags: {historic property, mixed zoning, undersized vacant, …}.

6.3.36. Rule Pattern: Explanation

Problem – Describing Sequence of Operation or Rule Invocations. Many activities in a business process may be performed simultaneously or in an arbitrary order because they are non-interfering actions. However, for audit purposes, the exact sequence of activities performed should be recorded. Such auditing may be the result of legal requirements, contractual obligations, or internal quality control. Nonetheless, some explanation facility is needed to report this information.

Solution. The auditing of activity or invocation of business rule enforcement is accomplished through process reification. The integration of adaptive use cases and parameterized business rules produce an executable model that can record the sequence of activities based on any or all of the view model stereotypes defined in Chapter five. There are two dimensions to the explanation facility. The first dimension addresses the scope of activities. The macro view looks at an entire business process and breaks it down to the desired level of detail. Most of the explanation facilities needed to support the stereotyped customer view of an adaptive use case model are provided in workflow engine based systems. The micro view simply examines a targeted action sequence of interest. When these actions are entirely automated, the facility must be built into the executable module. The second dimension addresses the nature of the explanation. An extensional view describes the activity sequence. This view explains what was done. Again, most workflow system will support reports of this nature. An intensional view describes the application of business rules that controlled the initiation of activities. These facilities are typically not included in automated systems unless a blackboard, knowledge based, or some other type of dynamic reflection architecture is chosen for implementation.

6.4. Business Rule Templates

A business rule template provides a mechanism for parameterized use of business rules. Just as a business rule pattern abstracts a collection of facts, a business rule template abstracts a collection of rules derived from a business rule pattern across types or sub-types. The business rule template shown in Example 6.52 demonstrates how a business rule pattern is sub-typed to create a family of business rules. In this example, the business rule pattern parameter set is combined with the business rule grammar syntax to first define the template in normalized business rule grammar form. The template is then converted into an alternate text representation. Next, the template is applied against a collection to bind it to a single business rule. Finally, the template is applied against several sub-types of the collection, resulting in three additional business rules at the meta-level. Each business rule subsequently ties to facts at the operational level (e.g. Russ Hurlbut may not have more than six library items checked out at any time).

Normalized Basic Template	Event: When changed – number of [libraryItemKind] checked out
	Condition: If a [PatronKind] has less than [libraryItemKindCheckedOutMax] [libraryItemKind] checked out
	Action: [PatronKind] checks out [libraryItemKind] from library
Basic Template	A [PatronKind] may not have more than [libraryItemKindCheckedOutMax] [libraryItemKind] checked out at any time.
Multiplicity Pattern	A patron may not have more than [libraryItemCheckedOutMax] library items checked out at any time.
	1. A patron may not have more than six library items checked out at any time.
Multiplicity Pattern with Sub-typing	A patron may not have more than [libraryItemKindCheckedOutMax] [libraryItemKind] checked out at any time.
	1. A patron may not have more than five books checked out at any time.
	2. A patron may not have more than four magazines checked out at any time.
	3. A patron may not have more than three videos checked out at any time.

Example 6.52. Business Rule Template

6.4.1. Binding Business Rules to Adaptive Use Case Templates

A business rule model can support an agent-based workflow system through its ability to translate business rules into facts. As just shown, facts reside on two level: the domain level and the instance level.

When a business rule is enabled and executed, it can generate instances of types in order to create facts. It can also change the state of the types that the rule is related to.

The transaction, control, and protocol rule pattern groups form the bulk of workflow oriented rules that are used for process flow control. The driving linkage for business rules is their bindings to process definitions. Business rules are only indirectly related to classes through the realization of use cases.

Business rules utilize their own templates to create sets of scenarios that comprise a use case. These rules are bound to the adaptive use case templates at build time. Thus, several different parameterized data structure formats can be accommodated in a single use case, each one relating to one of the business rules formed from binding of subtypes to the business rule template.

This dual binding of sub-types to business rule templates and the resulting business rules to the use case template allows for both the declaration of where variability is needed and the scope of that variability. Referring again to Example 6.52, the business rule template is bound to four rules. Each rule may be decomposed into a conditional test and an action as shown in the normalized business rule form. Each condition will eventually map to its own condition attribute in the *VariantScenarioAction* model element. However, the adaptive use case template only contains a single *VariantScenarioAction* because it only manages the location in the process flow where the variability is required. The binding of types (e.g. patron, library item) creates the scope of the variability that results in four separate conditional tests.

It could be argued that the same action could take place and that all four of these conditional tests are merely employing the *Association* business rule pattern. This would result in the four conditions being associated through disjunctive *or* clauses, resulting in a single derived attribute. The difficulty with this approach is that the individual characteristics of sub-types are obscured. Different types of patrons may have different limits or even be subject to different rules, such as an extended loan period or special consideration if overdue books are outstanding. Since the same *UseCaseSegment* model element can be imported by each condition, the only unique construct is the actual business rule that becomes attached to the condition attribute of a *VariantScenarioAction* as a result of its sub-type and parameter bindings.

Either approach will provide a solution at their respective extreme end of behavior variability. A single *UseCaseSegment* utilizing the *Association* business rule pattern consolidates all conditions to the same behavior. This works if variability is restricted to one or only a few parameters. At the other end of the variability spectrum, separate *UseCaseSegments* for each business rule condition that result from the business rule template allows completely different conditional tests to be introduced into *UseCaseSegments*.

A mix of the two extremes occurs through the use of parameterized states of an entity type that consolidates several conditional tests from different rules. Each of these different rules may be a result of binding from their own unique business rule template. Thus, it is possible to have a web of rules that can be clustered into rule families that drive the actual creation of *UseCaseSegments.* Extending our example one more time, we might have to determine the sub-type of a patron by applying one or more business rule conditional tests. These classification rules determine which behavioral rules apply. Consider the following rule: "If patron is a tenured faculty member, then the chief librarian on duty can override restriction of maximum number of checked out items." The determination of tenured faculty status of a library patron may come from a derived attribute utilizing the *Production* business rule pattern. The behavioral aspect of this rule combines an *Authorization* rule pattern with the previously discussed *Multiplicity* Pattern. Just as a *VariantScenarioAction* describe deltas in processes, parameterized sub-types in business rules only need to describe the difference in policy. By applying business rule grammar alternative structuring and then decomposing the rules into their condition and action components, binding to adaptive use case templates can be completed.

In addition to reuse at the meta-level when structuring adaptive use case templates and business rule templates, it can also occur at the parameter binding level. This is possible through use of the *Priority* business rule pattern that allows default rules to be overridden within the rule family.

6.5. Discussion

The business rule model and related parameterized business rule patterns presented in this chapter represent the second key mechanism that has been developed from the domain engineering conceptual framework presented in chapter four. The business rule model integrates aspects from the Herbst and GUIDE rule meta-models and extends them to be compatible with the UML meta-models [UML97]. A business rule meta-model also integrates workflow activities with use case formalisms that provide a basis for formalizing business rules as part of the domain architecture specification. The comprehensive parameterized business rule pattern language permits system requirements to be expressed in terms of a reusable configuration language through parameter value bindings. This business rules pattern language subsumes the Silva and Ross classification schemes [Silv95], [Ross97]. These parameterized rules provide a powerful specification tool that also incorporates the rule authority [Nass91], language formalism [Fall93], [SC95] and formatted requirements [Fink88] approaches introduced in chapter three.

Chapter seven brings together the adaptive use case model from chapter five and parameterized business rule patterns developed in this chapter into an integrated process model for managing domain architecture evolution. This domain evolution architectural transformation (DEAT) model will describe the relationship between the adaptive use case model and the business rule model in the context of transformations that are performed. This chapter has provided the basis for business rules to be integrated into adaptive use case constructs through the binding of business rule templates. Chapter seven expands on this relationship by considering the impact of new system development on the domain architecture. Adaptive use cases are transformed into sequence diagrams and then into event-state matrices. Through cluster analysis of event-state combinations, new functionality can be expressed through code segmentation representations that are readily transformed back into business rules. Thus, the relationship between adaptive use cases and business rule patterns is not merely a one way transformation. Rather, they combine to form essential mechanisms in an iterative process model that address two primary challenges in managing the evolution of a business domain architecture.

The first challenge, as previously described in chapter one, is to bridge the cognitive gap between management and software engineers. This is accomplished through the deployment evaluation sub-model of the DEAT process model in chapter seven. Business rules provide important feedback in building the change costing model that is intended to guide management's decisions. The second challenge is applying domain knowledge, design principles, standards, and programming skills to generate an application and to evolve the framework in such a way that desirable properties are maintained. These properties are identified in the domain engineering conceptual model from chapter four and the changes are enacted through the domain normalization operations that are prescribed as a result of the use case formalism and transformation sub-model of the DEAT process model. Capturing new functional requirements in the form of parameterized business rules helps to ensure a more resilient domain architecture and improve its understandability.

CHAPTER VII

MANAGING DOMAIN EVOLUTION

A business domain architecture must anticipate the introduction of new applications in a manner that minimizes the impact of change. The domain model needs to strike a delicate balance between flexibility and complexity. The architecture must be flexible enough to accommodate the new application. However, the specification and configuration of components to support the new application must not be overly complex. When changes to the architecture are required, they should conform to the principles of domain normalization presented in chapter four. New and enhanced functionality should be incorporated into the domain architecture through operations that preserve desirable normalized characteristics of the underlying domain model.

Building upon the domain engineering conceptual framework, adaptive use cases, and parameterized business rule patterns from the previous chapters, this chapter introduces a methodology for managing the evolution of a business domain architecture. A transformation process model prescribes a collection of representations to assess the impact of change. Operations to effect those changes are then performed on the domain model in a manner that is compatible with domain normalization principles.

This chapter first provides an overview of the transformation modeling technique. A process model for business domain architecture transformations is described. Next, detailed descriptions of the configuration, fit assessment, exception analysis, and change costing models are provided. The various use case formalisms included in the process model are then discussed. The chapter then proceeds to demonstrate the application of an event-state cluster analysis and various use case transformations through an extended example.

7.1. Domain Evolution Architectural Transformation Process Model

The domain evolution architectural transformation (DEAT) process model assumes that a domain model has already been developed and that at least one system has been implemented from the architecture. The focus of the DEAT methodology is on maintenance of an existing architecture. The DEAT process model differentiates itself from other object-oriented design and domain engineering methodologies by emphasizing management and evolution of the business domain over its definition. By segregating reuse and maintenance of the domain architecture from system development, the DEAT process may be considered an adjunct to exiting methodologies instead of a competitor. System development may proceed in its normal prescribed manner with the DEAT process steps integrated at appropriate places. In a sense, these steps may be considered an extension to other methodologies in much the same manner in which a use case may be extended with additional behavior.

7.1.1. Deployment Evaluation and Domain Evolution Iterative Processes

Two iterative process characterize the DEAT model. The deployment evaluation process begins with the system requirements model, as shown in Figure 7.1. The systems requirements are matched with the domain model to generate the configuration model. The transformation process involves a visual programming interface to domain specific languages that evolve from the domain model. Based on the how closely the configuration model matches the system requirements, the fit assessment model is constructed to evaluate the unsatisfied requirements in the system requirements model. Adaptive use cases are considered in this assessment to provide a preliminary analysis of the impact of changes needed. The exception analysis model is then derived from the fit assessment model with additional input from use case formalisms. Feature coverage profiles from event-state matrices are integrated to prioritize delivery of features and handling of process breakdowns (i.e. exceptions to the typical course of actions). The change costing model then incorporates project management aspects with respect to resources of time, money, personnel, and development environment assets. The project plan and budget are then fed back into the system requirements model to reassess scope and feature delivery timetables.

The primary purpose of the deployment evaluation iterative process is to assess the robustness of the domain model and to quantify its contribution to the system development process. The second iterative process centers on evolution of the domain model through transformations of use case formalism. The purpose of these transformations is to refine the process, policy, and structural view of the domain model. Then, domain normalization operations may be performed to effectively accommodate changes needed for current and future application instantiations. By supporting the fit assessment iterative process, the use case formalisms directly contribute to the change costing model. Thus, costs may be allocated for domain evolution that is amortized over future application instantiations even though the changes are targeted for a specific application.

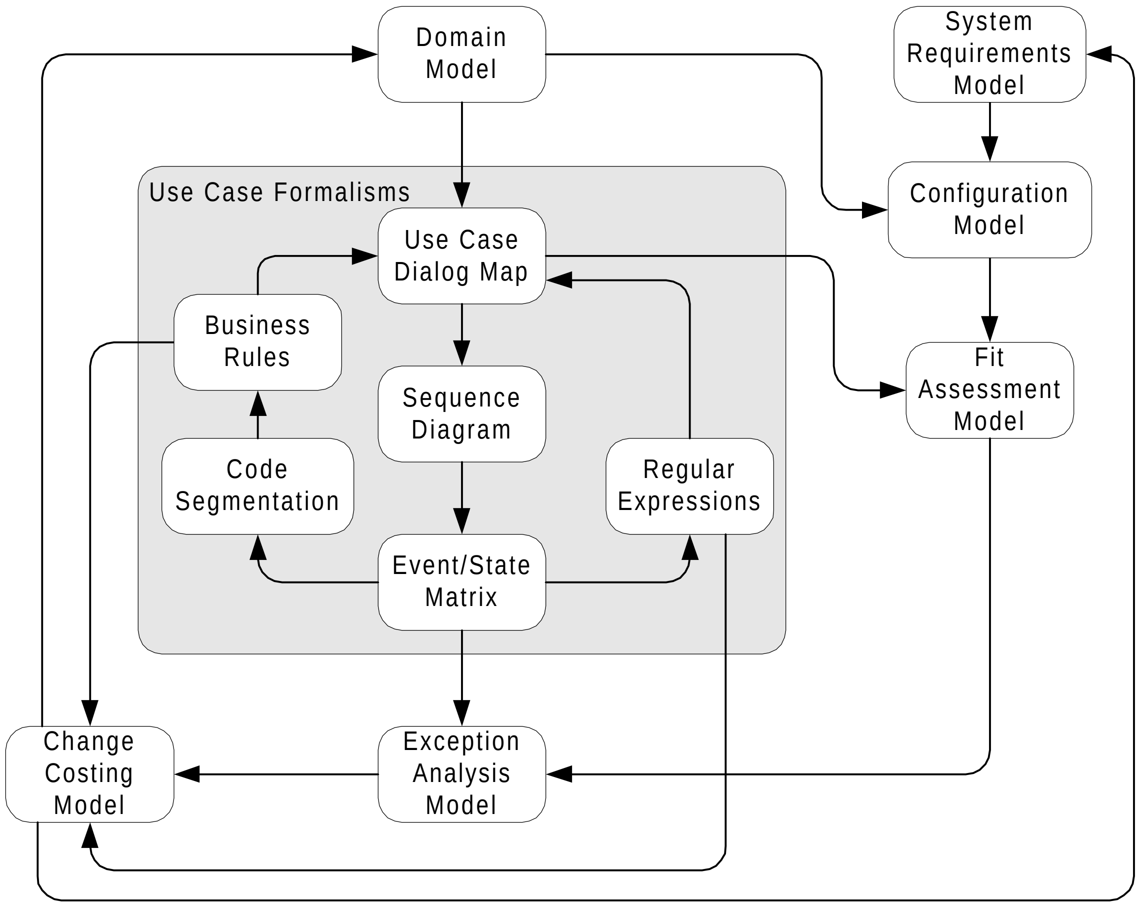


Figure 7.1. Evolutionary Architecture Transformation Process Model

7.1.2. Use Case Transformation Sub-Model

The use case transformation sub-model consists of the representations enclosed by the gray box in Figure 7.1. Of the formalisms presented, only the sequence diagram is formally defined in the Unified Modeling Language (UML) notation guide. The event-state matrix is indirectly considered as a constrained version of a statechart, of which the UML *StateMachine* is an object variant. Two additional formalisms are included in the transformation model – use case regular expressions and code segmentation. These two representations serve a similar purpose. Both contribute to development of the change costing model and provide a mechanism for verifying and validating the adaptive use case.

As new functionality is introduced through definition of new use cases or additional scenarios to existing use cases, they are mapped to existing components via transformation to a sequence diagram. This is a straightforward mapping that is defined in the UML semantics document and notation guide. The sequence diagram is then analyzed for new component level interactions by identifying stereotyped object roles.

The event-state matrix provides a protocol specification showing the order in which operations may be invoked on the system by the actor. A minimization algorithm may be applied to the event-state matrix that yields clusters of similar target states. A cluster analysis is then performed to identify component boundaries. The resulting normalized event-state matrix enables mapping to code segmentation and regular expression representations. Moreover, the normalized matrix facilitates a feature coverage analysis. Such an analysis contributes to creation of the exception analysis model.

Although actions are not directly shown in the event-state matrix, they become manifest in the code segmentation representation that consists of event-condition-action case scenarios. These code segments are constructed in a manner that permits direct mapping to business rules. Through application of the business rule patterns presented in chapter six, an assessment may be made as to how the new functionality should be integrated into the adaptive use case model. If parameterization already exists for this particular rule, then the only impact is on the configuration model. The change costing model and adaptive use case model will be unaffected. On the other hand, if a new type of business rule is revealed or if additional variability is needed in a matching business rule pattern, then the necessary changes will be recorded in the change costing model based on the impact to the adaptive use case model.

Regular expressions provide an alternate representation of the behavior life cycle of entities captured in the event-state matrix. When combined, these expressions represent the essential dialog that exists between an actor and the system. They provide a vehicle for validating normalized usage of any system derived from the domain model. Validated usage dialogs may then be migrated back into the adaptive use cases.

Feedback to the adaptive use case model from regular expression and business rule refinements may trigger another iteration of the transformation process. Changes to an adaptive use case are verified against the structural view through the sequence diagram, which in turn leads to further transformations if changes are needed in that view. The iterative cycle ceases once accounting for all modifications has concluded.

7.2. Deployment Evaluation Models

7.2.1. Domain Model

Although initial development of the domain model is outside the scope of the DEAT methodology, several artifacts are expected to be present as part of the model. A concept map that defines specific entities and their interactions should be provided to assist in transforming the system requirements model into the configuration model. Also, a release plan of functionality should reveal the strategy for deployment of multiple products from the domain architecture. Specifically, it should identify the variability spots of the architecture where additional component are planned. Identification of potential locations for opportunistic parameterization should reveal how the architecture is expected to mature. Targets for evolution of the architecture in terms of the defined attributes for the domain architecture, such as coverage, stability, and efficiency, should be established along with related milestones and priorities.

7.2.2. System Requirements Model

The systems requirements model is constructed from the consolidation of two initiatives. First, it represents the customer's prepared document of functional and non-functional requirements for the system to be developed from the domain model. Second, it represents compromises and tradeoffs of functionality based of the capabilities of the domain model. Progressive refinements are applied as a result of elaboration of the downstream models as shown in Figure 7.1. The preferred structure for the model is a structured workbook format that closely parallels the configuration model. This facilitates traceability from the domain model architecture to each system requirement identified.

7.2.3. Configuration Model

The configuration model is derived from the domain model and system requirements model. It inventories and analyzes the system requirements by comparing calculation elements against the existing parameterized business rules in the domain model. Components are selected from the domain model for those elements that are compatible with the new system requirements. A visual programming interface may provide graphical constructs to pick and choose exemplars (previously defined parameter blocks) and import them into the current configuration specification. The resulting preliminary configuration model serves as a starting point for assembling the system from the domain architecture. It is possible for some exemplars to include the entire configuration model of a previous system implementation. Any exemplar that contains a composite of parameter blocks may be used as a base and then modified incrementally. For either intended usage, many of the reused components from the domain architecture will have new parameter values assigned to precisely configure the system.

When parameter value assignment is insufficient to match system requirements, existing components are used as a starting point to extend or modify the application logic. During this step, any functionality that represents new types of business rules is identified. At each spot in the domain architecture where a pluggable system requirement is needed, the task of selecting the applicable requirement component is performed by a *requirement coordinator*. The requirement coordinator is realized by an object in the domain architecture and conforms to one of the factory design patterns [GHJV94]. If there is more than one possible component, a determination is needed as to which one is applicable for a given context. It is also possible that more than one can apply in any context. Once the correct set of one or more requirements has been determined, the correct process component is instantiated with appropriate values.

The component selection and value binding for application deployment is accomplished entirely through parameters. This recurring usage of parameter value bindings has led to the development of the *Parameterized Configuration* design pattern. Two examples of the application of this pattern are shown in Figure 7.2 and Figure 7.3. This design pattern combines a *Composite* design pattern [GHJV94] with the *Range Set* and *Domain* parameterized business rule patterns. The pattern is characterized by the layered sets of parameter values that realize the *Domain* business rules through the *Composite* design pattern. The context for parameter selection is then established through a collection of the lookup objects that realize the *Range Set* business rules. The *Parameterized Configuration* pattern captures system requirements without regard for actual deployment options, such as binding time and physical location. The parameters may be bound to the components at build-time or dynamically at run-time. The deployment location can be in the presentation, application, or persistence layer of the application.

Hybrid deployment strategies that balance tradeoffs between flexibility and performance are also possible. For example, a graphical user interface (GUI) may require dynamic changes to validation rules for input forms. A run-time strategy can be adopted to store parameter values in a relational database. The tables may be denormalized to remove the recursion inherent in the configuration pattern's logical structure, as shown in Figure 7.2. In this manner, the nature of the persistence layer is leveraged to sacrifice performance on infrequent changes to edit validation rules in order to improve overall performance of the main course of action involving the input forms. Thus, the database stores the full collection of tables that capture the semantics of the business rules with respect to overridden default values, but caches the visible parameter values in a separate table. The cached table is automatically regenerated though a transformation algorithm based on the derived attribute *editValidationRules*. The formal semantics are expressed in Object Constraint Language (OCL) syntax of the UML as follows:

[1] The *getRules* operation of an *EditControlRule* returns itself as the sole element of a set.

```
EditControlRule::getRules() : Set {EditControlRule}
post: result = Set {self}
```

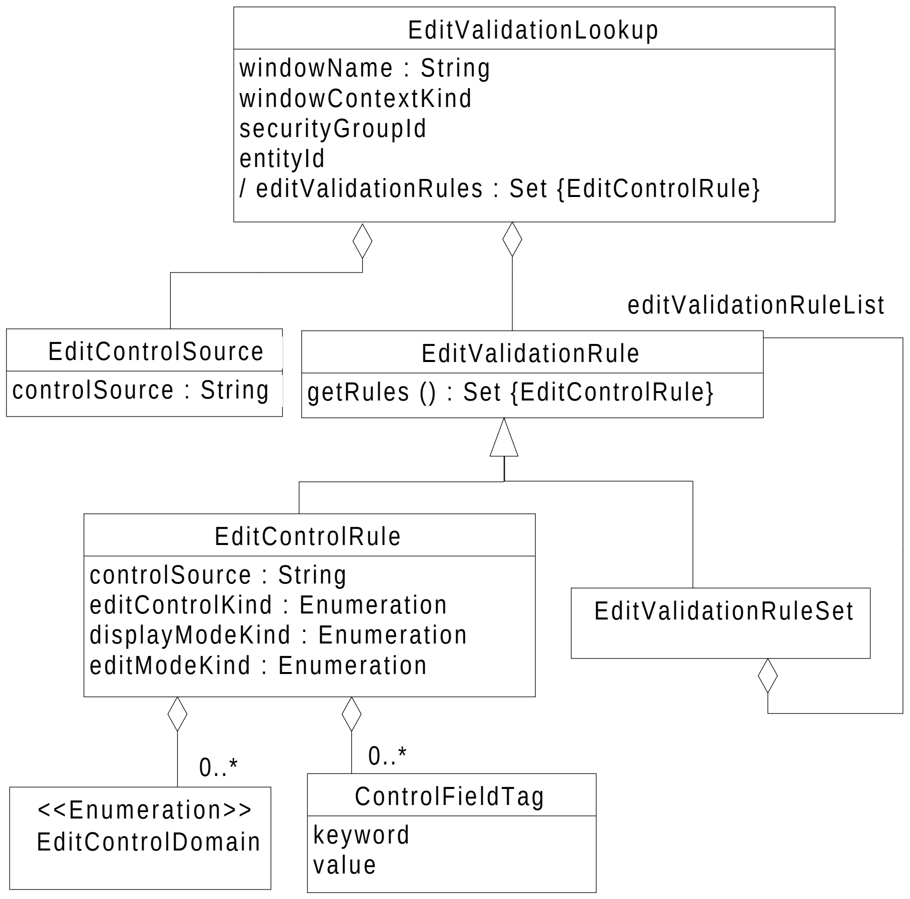


Figure 7.2. Presentation Logic Configuration Model

[2] The *getRules* operation of an *EditValidationRuleSet* returns each of the *EditControlRules* that are directly contained in its *editValidationRuleList* attribute. The return set also contains all *EditControlRules* recursively contained in *EditValidationRuleSets* that have *controlSources* different from the directly contained *EditControlRules*. An additional operation, *getTopLevelControlRules*, is defined to clarify the expression.

```
EditValidationRuleSet::getRules() : Set {EditControlRule}
post: result = self.getTopLevelControlRules->union (
                  self.editValidationRuleList->reject ( rs |
                        self.getTopLevelControlRules->exists (
                              controlSource = rs.controlSource )))


getTopLevelControlRules (): Set {EditControlRule}
getTopLevelControlRules = self.editValidationRuleList->select ( r |
      r.oclIsTypeOf ( EditControlRule ) )
```

[3] The derived attribute *editValidationRules* generates a set of *EditControlRules* by invoking *getRules* on each member of its editValidationRule attribute. The returned list is filtered by the controlSources contained in its editControlSourceList attribute.

```
EditValidationLookup::editValidationRules : Set {EditControlRule}
post: result = (Set{}->union( self.editValidationRuleList->forAll ( r |
            r.getRules)))->select (r1 |
                  self.controlSourceList->exists (
                        controlSource = r1.controlSource))
```

The various enumeration types of attributes in the presentation logic configuration model provide precise control over the display characteristics of an input form. The *securityGroupId* conforms to the *Authorization* business rule. It combines with the *windowContextKind* and *entityId* to establish an overall context for the input form. The *windowContextKind* attribute establishes data context such as current values, historical values, or edited values pending review. The *entityId* associates the form within a business context such as a real estate property type, insurance carrier, or school curriculum. By establishing multiple contexts through parameterized business rules, a series of default edit control properties can be shared. Only values that require precise control need to be overridden through parameter specification.

At the edit control level, the *editControlKind* attribute identifies the type of control, such as textbox, list, choice, and button. The *DisplayModeKind* attribute contains enumerated values such as muted, normal, and highlighted. The *editModeKind* attribute describes editability characteristics such as read only, editable, and editable with peer review. The control source identifies the data attribute that is bound to the control. If it is an enumerated domain, the attached *editControlDomain* list is utilized to describe its value based on declarations made in a parameterized *Domain* business rule pattern. Other domain constraints, such as minimum and maximum values may be customized through keyword-value pairs that are built into the architecture.

Through normalization of the parameter values, control rule attributes are typically shared across many contexts. A simple factoring of shared values to the most global default level would provide the highest degree on normalization. However, semantic differences in values that just happen to currently be identical must be captured. For example, an application may set a default rounding value to four decimal places. A few of these calculations may be subject to contractual obligations that are also set to four decimal places. If the terms of the contract are renegotiated or the organization decides to modify its default value, the impact of change should only be propagated to appropriate places without undesirable side effects. Therefore, the configuration model must identify these semantic differences and capture them via parameterized business rules that trace back to the system requirements model.

The application logic configuration model in Figure 7.3 is constructed to address such semantic differences. Thus, the presentation and application sub-models work in tandem to fully specify the configuration model. Variants that combine aspects of both sub-models may be used for specific needs, such as interfaces and type conversions between systems. Attribute and collection business rule patterns that share edit validation rule parameters may also share system parameters through collaborations with entities specified in transaction business rule patterns for exception handling.

The formal semantics to generate the parameters for the application logic model are identical to the presentation logic model. Since only the names of the model elements are changed, the OCL expressions will not be repeated. The primary difference occurs in the replacement of the lookup object with another *Composite* pattern. This generalizes the selection of the appropriate context. The three attributes that determined context for the presentation logic were specifically designed for the application of the pattern for GUI interaction. The additional layer of variability demonstrates a recursive application of the *Configuration Parameters* pattern on itself. Other differences between the two models lie in the declared attributes and subtypes representing parameter types declared in the configuration model.

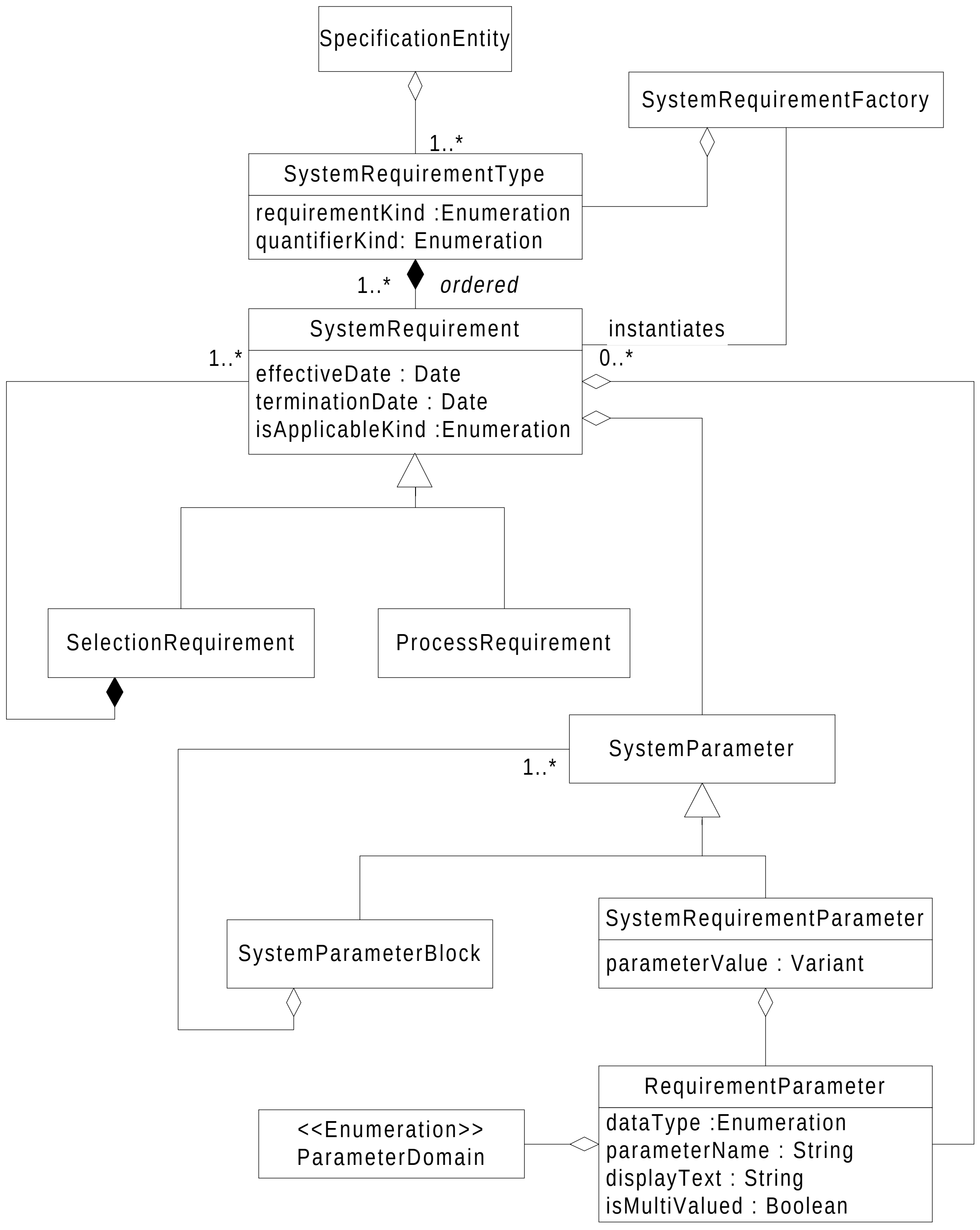


Figure 7.3. Application Logic Configuration Model

Quantifier Parameter. The model is intended to support two types of quantifier parameters: those that conform to the *Association* business rule pattern and those that conform to the *Aggregation* business rule pattern. The *Association* parameter values are *any, all,* and *each*. They are used when the parameterized process is intended to evaluate to a Boolean. The *Aggregation* parameter values may be any appropriate aggregate property, however *minimum* and *maximum* would be most commonly used. If there is only one requirement object for a particular type, the quantifier has no immediate effect. When there is more than one requirement, then this parameter indicates when processing ceases.

If *SystemRequirementType.quantifierKind* is specified with a value of *any*, the first requirement object that is applicable is considered. Since the collection of system requirements is ordered, the selection is made in a deterministic manner by applying the *Priority* business rule pattern to establish the ordering. The applicability of a requirement is usually determined by selection parameters. If *all* is specified, then all applicable requirement objects are considered and must evaluate true. Usage of *each* preserves all side effects regardless of any Boolean evaluation. When a value other than a Boolean is returned as the result of the parameterized process the *Aggregation* parameter values are appropriate. In this case, each applicable requirement is considered and the collection of return values is used to apply against the business rule pattern.

Selection Parameters. Selection parameters determine when a particular system requirement is used. It applies the *Type Membership* business rule pattern to classify certain system requirements as applicable only in certain contexts. The *SystemRequirement.isApplicableKind* domain will vary with each type of system requirement. When the quantifier parameter is specified, those that return a value of the appropriate type will be classified with the same type membership. The set of applicable system requirements is then determined by evaluating membership according to the following OCL expression:

```
SystemRequirementType::
getApplicableRequirements : Sequence{SystemRequirement}
post: result = systemRequirement->select( r |
      r.isApplicable.oclIsKindOf(self.quantifierKind.oclIsTypeOf))
```

When selection criteria is more complex than this basic mechanism, a selection requirement provides the control logic to perform evaluation of applicability. Selection requirements may also invoke additional selection requirements when the criteria are sufficiently complex. In most cases application of the *Range Set* business rule pattern, such as that used in the presentation logic model, should suffice.

Time Sensitive Parameters. Any or all system requirements may be time sensitive. For example, one system requirement may be used for a specific period of time, but then another is used after that. Evaluation of time sensitive requirements requires binding to a date-time attribute. The identification of this data attribute may be specified as a system parameter. In this manner, the additional layer of instantiating a selection requirement may be avoided.

Two basic types of time sensitivity may be exercised by declaration of the effective and termination date attributes associated with the system requirement. The first deals with a point in time that is identified by one of the system requirement parameters. The second deals with a range. This second type is suited for declaration of an aggregation type of quantifier, such as *sum.* In this case, each applicable system requirement is applied during its time segment.

Process Parameters. Process parameters allow for different methods or strategies to be invoked. When alternate calculations are performed in sufficiently disparate ways, the process used along with the corresponding set of parameters may be quite different. Each process parameter has a unique corresponding object class in the domain architecture that conforms to the interface type associated with the system requirement type. These are the objects that get instantiated by the system requirement factory.

Local Control Parameters. Local control parameters provide flexible behavior within the class identified by the process parameter. Therefore, the process parameter is always identified before the values for local control parameters have any relevance. Such parameters specify whether or not certain actions should be taken within the process. For example, a calculation may be derived from two table lookups, the first for an intermediate value and the other for the final value. An execution parameter may specify whether or not intermediate rounding should be performed as well as how it should be performed. These types of execution parameters allow the class to be more generalized. The distinction between a process parameter and a local control parameter is inherently bound to implementation of the domain architecture. When configuring a system through parameter specification, this distinction should not be apparent to the user.

Calculation Parameters. Calculation parameters are the simplest type of parameter both conceptually and with respect to implementation. Typical usage of a calculation parameter is to identify a threshold. For example, a projected salary calculation may set an annual increase percentage to five percent. This value would be the 'projected salary increase' execution parameter. As with local control parameters, the process parameter must be identified before the values for calculation parameters have any relevance.

Construction Parameters. Construction parameters deal with non-functional issues, such as build-time versus run-time binding of parameters. They may be used in a context that reflects the life-cycle state of the application. For example, if the application is in a prototyping stage, run-time binding may be preferred to permit rapid consideration of many configuration options. Once the application logic has been stabilized, a move to build-time binding of parameters may be preferred to improve performance and prohibit undesired changes.

7.2.4. Fit Assessment Model

The fit assessment model is intended to provide non-technical users of the domain architecture with a mechanism to evaluate the appropriateness of a framework. The outcome of parameter selection for the configuration model is used to drive creation of the fit assessment model. It is possible that many parameter values are unknown after a preliminary effort to construct the configuration model. This may occur in spite of a relatively complete matching of components to requirements. The fit assessment model summarizes the effort required to complete the application. There are three facets to this assessment.

The first facet assesses the fit of the system requirements model to the domain specific parameterization language. An evaluation of the configuration model quantifies how closely the parameter match the functional requirements. If components exist to address all functional requirements, then the programming effort is restricted to this higher-level configuration language. The resulting metrics provides an estimate of the remaining programming effort in the configuration language.

The second facet assesses the fit of the fulfilled functionality from the system requirements model to the domain model. This assessment provides an estimate of the remaining requirements engineering effort to determine unknown parameter values through interaction with the customer. Gaps often occur in the system requirements model due to under-specification. Therefore, this facet may also be considered as a refinement iteration of the system requirements model.

The third facet assesses the fit of unfulfilled functionality from the system requirements model to the domain model. Unfulfilled requirements not fulfilled by the domain architecture will require additional component development and modifications to address new functional requirements beyond the current scope of the domain architecture.

The three assessment facets are quantified through evaluation of the current state of the configuration model. The application logic configuration sub-model provides a measurement of the selection and process requirement complexity. Presentation requirements are measured in terms of their complexity in constructing the presentation logic sub-model. The persistence and interface requirements provide additional measurements when needed to support these first two configuration sub-models. Figure 7.4 shows the relationships between the component parts of each model.

<u>Process Requirement Complexity Levels.</u> The nine levels of process requirement complexity are described in Table 7.1. The simplest implementation will exclusively utilize components from the domain architecture in previously configured ways. This will only require selecting the appropriate configuration parameter blocks. The second through fourth levels add additional complexity since the existing components will be used in different contexts than in the past.

Levels five through nine introduce domain model changes. Level five injects variability where a constant previously existed. Level six adds complexity through additional control flow logic within the component. These two levels are realized through the *Extension* type of domain normalization operation. Level seven requires development of a new pluggable component. It is realized through the *Completion* domain normalization operation. Level eight involves modification to the template, or static portion of the domain architecture. The creation of a new variable spot in the domain architecture is performed through a *Stratification* domain normalization operation. The ninth and final level introduces an entire new subsystem. This typically also involves significant modification to the static portion of the domain architecture to accommodate this new functionality. The *Integration* domain normalization operation is used is such a situation.

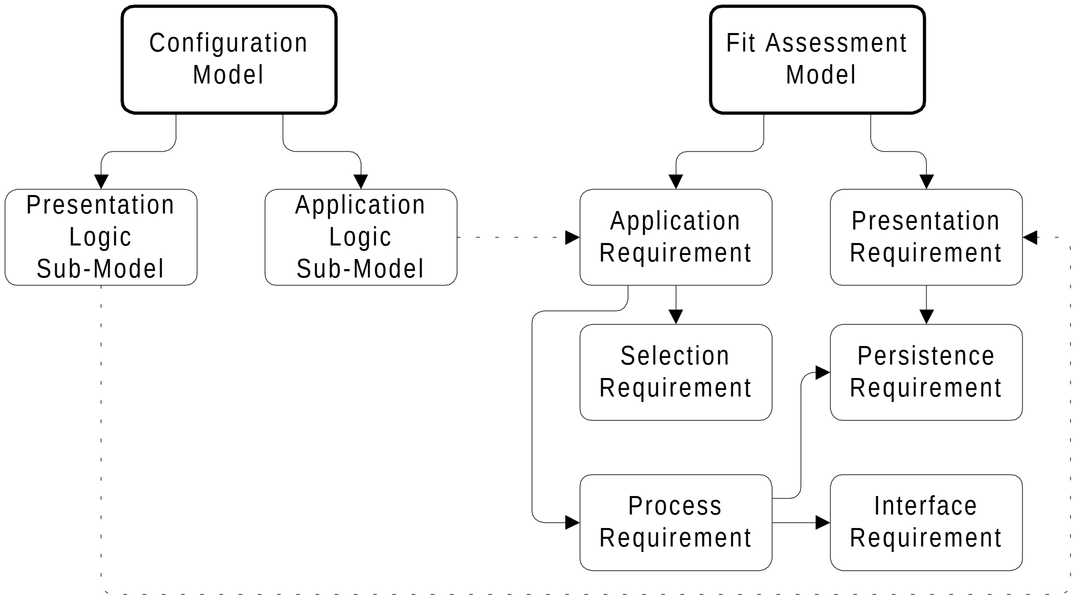


Figure 7.4. Mapping of Configuration Model to Fit Assessment Model

Table 7.1. Process Requirement Complexity Levels

Level	Affected Model	Characterization
1	Configuration	Complete parameter value reuse
2	Configuration	New calculation parameter values
3	Configuration	New local control parameter values
4	Configuration	New process parameter combinations
5	Domain	New calculation parameter types
6	Domain	New control parameter types
7	Domain	New process
8	Domain	New process type
9	Domain	New subsystem

Selection Requirement Complexity Levels. The six levels of selection requirement complexity are described in Table 7.2. The first three levels only involve parameter specification. Level two reorganizes the clustering of process requirements. Level three reorganizes the selection requirements themselves in new configurations. The remaining three levels all involve domain model changes. Level four utilizes a state-based lookup to determine which process component gets instantiated. Criteria that conforms to the

Range Set business rule pattern will be easier to specify than other, more complex conditional logic. Although it represents the same type of *Extension* domain normalization operation as level five, the pattern based specification reduces the overall complexity. The final level introduces multi-tiers selection logic involving recursive instantiations of one or more additional selection requirement components.

Table 7.2. Selection Requirement Complexity Levels

Level	Affected Model	Characterization
1	Configuration	Complete parameter value reuse
2	Configuration	New quantifier/applicable sub-types
3	Configuration	New selection requirement combinations
4	Domain	New state based criteria – Range Set Pattern
5	Domain	New selection requirement
6	Domain	Recursive selection requirements

<u>Presentation Requirement Complexity Levels.</u> The seven levels of presentation requirement complexity are described in Table 7.3. These levels capture specialized structure for application logic that is presented at automated system boundaries including GUI screens, voice response systems, reports, and other structured communication materials, such as forms, letters, brochures, and advertisements. The first three levels limit complexity to parameterization. Level two introduces new parameter values, such as minimum and maximum value thresholds. Level three defines new enumeration domains. The fourth level requires modification to presentation logic to parse new tag types that deal with unique validation requirements of attribute values. Level five defines new controls, which is usually associated with a new attribute in a persistence requirement. New presentation formats introduce design of presentation formats such as GUI forms, voice response scripts, and letters. This level represents a *Reconstruction* domain normalization operation. Finally, a new presentation type would define a deployment option, such as the introduction of bar coding for input and output.

<u>Persistence and Interface Requirement Complexity Levels.</u> The persistence and interface requirements share the same domain for fit assessment complexity. These levels are described in Table 7.4. Level two involves specification for type conversion for existing attributes. The remaining two levels

introduce new attributes and new entities, respectively. The introduction of these new constructs is performed in conjunction with application and presentation requirements. The distinction is made here as a separate requirement type in order the further recognize familiar aspects to domain evolution that have a well understood impact on the adaptive use case model and other use case formalisms.

Table 7.3. Presentation Requirement Complexity Levels

Level	Affected Model	Characterization
1	Configuration	Complete parameter value reuse
2	Configuration	New parameter values
3	Configuration	New domain types
4	Domain	New tag types
5	Domain	New controls
6	Domain	New presentation format
7	Domain	New presentation type

Table 7.4. Persistence Requirement Complexity Levels

Level	Affected Model	Characterization
1	Configuration	Complete reuse
2	Domain	New attribute type
3	Domain	New attribute
4	Domain	New entity

Having established the complexity levels of the various requirement classification types, formal presentation of several fit assessment measurements is possible:

System Requirement Completeness. A measure of fit through fulfilled functionality from the system requirements model to the domain model is expressed as the percentage of fully specified requirement blocks for components with respect to total parameterized components that have been specified. This measure provides an indication of the completeness of the system requirements model that may be extrapolated to unspecified components that must be developed or extended.

$$SRC = \frac{F}{R} \tag{7-1}$$

Where:	SRC	=	System requirement model completeness
	F	=	Fully specified requirement blocks for components
	R	=	Total parameterized component that have been specified

Parameter Coverage. The parameter coverage metric provides a more precise measurement of fulfilled functionality by calculating the amount domain specific language completed. It represents the average number of specified parameters for each parameterized component identified. This value will always be greater than or equal to the system requirement model completeness metric since they are included in this calculation. This measure provides an estimate of the percentage of new and extended functionality that will be able to be assigned parameter values. Thus, it indicates the degree of refinement still needed in the system requirements model.

$$PC = SRC + ((1 - SRC) \times \sum \frac{f}{r}) \tag{7-2}$$

Where:	PC	=	Parameter coverage
	SC	=	System requirement model completeness
	f	=	Specified parameters in a single parameterized component
	r	=	Total parameters in a single parameterized component

Domain Model Coverage. Domain model coverage provides an approximation of the fulfilled functionality from the system requirements model to the domain model. It is defined as the percentage of components that can be configured with parameterization only. A separate measure may be provided for each requirement classification type as well as a composite of all requirements. It provides an indication the ability of the domain model to meet the needs of system development. Moreover, this metric provides an estimate of the remaining programming effort in the configuration language once the new and extended functionality has been implemented.

$$DC = \frac{D}{T} \tag{7-3}$$

Where:	DC	=	Domain model coverage
	D	=	Component that can be used with parameter configuration only
	T	=	Total components including new and extended ones

Domain Model Enhancement. The domain model enhancement metric provides an approximation of the unfulfilled functionality from the system requirements model to the domain model. It is defined as sum of all components identified as new development or expanded functionality. A weighted factor is applied to each complexity level to provide a relative rating among the defined levels for each respective requirement type. Just as for the domain model coverage metric, this calculation may be performed for each requirement classification type. The composite measure is the sum of all requirement types.

$$DE = \sum (c \times f(dl)) \qquad (7\text{-}4)$$

Where:	*DE*	=	Domain model enhancement
	c	=	Components that are expanded or integrated
	Dl	=	Complexity level for component
	f(dl)	=	Function that assigns a relative weighting to a complexity level

7.2.5. Exception Analysis Model

The exception analysis model (Figure 7.5) is intended to provide an intermediate representation for combining estimates of remaining effort to complete a system application from the fit assessment model and the potential new features that may be added as a result of event-state matrix normalization. This model provides a "framework for discussion" by serving as a mechanism to quantify the dialog between development and the client. A developer may point out places revealed during the event-state matrix normalization where potential risk of system breakdowns in processing is present. The client must then consider the cost and whether or not he is willing to pay for them. The exception analysis also addresses how to factor specific client needs versus expansion of the conceptual domain model which will allow amortization of cost over future projects/applications.

As part of the exception analysis, each potential new feature discovered should include an assessment these facets:

Importance. The determination of relative importance of the discovered feature will reveal if this was an oversight when specifying the systems requirements.

Probability. The likelihood of this situation occurring within the current system context may be non-existent. However, the generalized behavior of the domain architecture may need to deal with such an occurrence in a broader context.

Risk. The ramifications of not providing a response if this situation does occur must be assessed with respect to any resulting integrity violations and invalid entity states.

Marginal Effort. Since the discovery of the feature arose out of analysis for expanding existing components or integrating new ones, an assessment should be made for the additional effort to include this feature at the same time as the base changes. A comparison of the effort to include at the same time versus deferring inclusion is also a part of this assessment.

Response. Two business rule patterns are specifically constructed for addressing exceptions and breakdowns in normal processing. The application of the *Exception* business rule pattern will specify the conditions for occurrence of the situation. The *Enforcement* business rule pattern specifies how that response will be performed.

Model Allocation. Based on risk, probability, and other pertinent factors, an assessment should be made as to how the allocation of effort should be distributed. If there is no relevance within the context of the current system being developed, the entire allocation should be to the domain model. When functionality is so specialized that its reuse potential is non-existent, the allocation should be entirely placed on the system requirements model.

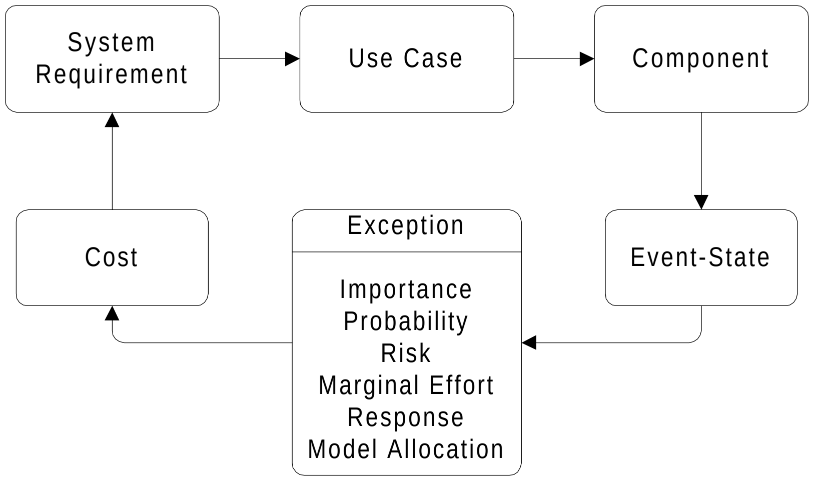


Figure 7.5. Exception Analysis Model

7.2.6. Change Costing Model

The change costing model integrates the output of the exception analysis with business rules and formal use case regular expressions. It provides an interface to the project management resources of time, money, personnel, and development environment assets. The project plan and budget are then evaluated against the system requirements model to reassess scope and feature delivery timetables. Model allocations of changes from the exception analysis also interact with use case formalisms to allocate resources for domain evolution. These costs are amortized over future application instantiations even though the changes are targeted for a specific application. By applying rules through application of the *Enforcement* and *Exception* business rule patterns, the change costing model also serves as a decision model for defining boundaries among automated, semi-automated, and manual processes.

The change costing model is a subset of the total project management picture for system development. It recognizes the domain evolution costs as separate and distinct from system development costs. If parameterization already exists for this particular business rule, then the change costing model is unaffected. New types of business rule or the recognized need for additional variability in a matching business rule pattern is recorded in the change costing model based its the impact on the adaptive use case model. As discussed in the description of the exception analysis model, the decision must be made as to whether or not additional costs are justified.

Total cost of change may be calculated as a function of the domain model enhancements that are allocated for the components to support the featured functionality, the parameter coverage, and the model allocation factor. This calculation applies the Shanley principle [Jack94], by recognizing that costs may be allocated to components that are utilized by two or more feature enhancements. Repeated application of this calculation for different combinations of feature sets provides a means for comparison and prioritization.

$$CC = DE \times \frac{1}{PC} \times MA \tag{7-5}$$

Where:	CC	=	Change cost
	DE	=	Domain model enhancement
	PC	=	Parameter coverage
	MA	=	Model allocation

Budgetary constraints usually preclude consideration of all potential features. By associating component functionality with normalized use case actions, the fit assessment level assignment may be quantified through unique use case action construct that are created to be implemented by the components in a manner similar to use case points [Hans95] and change cases [EDF96]. The change costing model only prescribes the method of computation for assessing total cost of anticipated changes. Either of these related techniques may be integrated into the model as the means to quantify the relative weight of the fit assessment levels.

The cost curve of building and maintaining a domain specific architecture can be considered by applying the change costing model over a series of system implementations. From an economic point of view, the cost curve of evolving a domain specific architecture should be analyzed. At some point, the flexibility may far outweigh the cost of "one at a time" development. This situation often results from an occurrence of the *lathe of heaven* syndrome[3]. This syndrome reveals itself when several different grandiose plans are only partially implemented. This may be for any number of reasons, such as the introduction of a new technology or replacement of a chief architect. Nonetheless, it results in increased complexity and reduced understandability. The *lathe of heaven* syndrome can cripple the development environment unless domain specific tools are simultaneously developed to manage complexity. There are certain patterns that can be identified in which the basis for a generic visual programming language can be established. In other words, we implement visual devices to manage the transition to an environment with domain specific constructs that share a common abstraction across any domain – such as enumerating values for a domain as expressed in a business rule pattern. The costs of developing these domain specific abstractions must also be considered in the weighted factor for each planned domain model change.

7.3. Use Case Formalisms and Transformations

Transformations of use case formalisms permit consideration of the underlying domain model through various partitioning techniques as shown in Table 7.5. Each representation targets a specific aspect of the domain model to partition in order to reveal how entities interact.

Table 7.5. Use Case Formalism Partitioning Techniques

Type or representation	What is partitioned	How objects interact
Event flow diagrams	Components	Object types
Event-state matrices	Code segments	Behavioral lifecycles
Use case dialog maps	Transactions	Dialog patterns

[3] The lathe of heaven syndrome is an expression derived from an Ursula LeGuin science fiction novel. "The city, half wrecked and half transformed, a jumble and mess of grandiose plans and incomplete memories swarmed like Bedlam..."–LeGuin, U. K., The Lathe of Heaven . Avon Books, New York, 1971.

With the convergence of modeling constructs through the use UML meta-models, direct mapping from one representation to another has now merely become a matter of providing a mapping to the model elements contained in the UML specification. The transformation then becomes a two step process – first to the UML meta-model and second to the target representation, as shown in Figure 7.6. Thus, it is the process of when to make a transformation from one representation to another that gives the DEAT process its distinctive nature rather than how the transformations are performed. The manipulation of each representation through the application of domain engineering normalization operations provides visual cues as to the next logical transformation that should be considered.

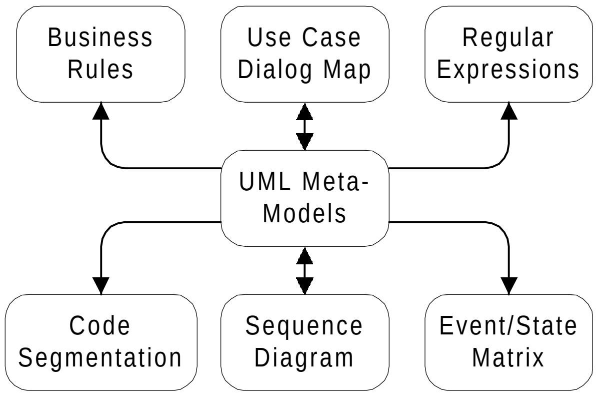


Figure 7.6. Transformations Using UML as Underlying Model

7.3.1. Use Case Dialog Maps

Use case dialog maps (UCDMs) directly reflect the elements in the adaptive use case model as describe in chapter five. This includes all alternate paths and exception handling. If changes are made to the use case dialog map as a result of transformation cycles of use case formalisms, the last iteration should originate from a use case regular expression in order to validate normalization. In a normalized use case, all commonality is factored out through shared use case actions. A single step does not usually constitute an appropriate unit. Since use cases are comprised of alternate paths and transactions, a natural two-dimensional partitioning of use cases already exists. Any transaction can have several alternate paths. It may also be possible for the alternate path to appear in more than one transaction. This is captured in the use case dialog map as a *transaction point.* A *transaction point* occurs in each use case action where two or more use case actions can converge or when subsequent alternative use cases actions are possible. These *transaction points* define boundaries and interfaces for recognizable decompositions of use cases.

Use case dialog maps are useful in analyzing relationships *within* use cases. In order to analyze the relationships *between* use cases, we attempt to recognize the commonality of use case segments across use cases. When looking at features of normalized use cases, complexity, performance constraints, frequency, and importance all need to be considered. These factors all contribute to a normalization of a family of use cases. The *transaction points* identify where appropriate abstractions can be "spliced in", such as in a *uses* scenario. The ability to use UCDMs permits creation of special domain specific dialogs, such as for handling exceptions where manual intervention is needed.

One of the desired objectives of the Use Case Dialog Maps is to allow the regular expressions to be depicted in a more visual representation. It becomes easier to identify what are the rules and what are the exceptions. This can be done through the visualization of alternate paths through the use case scenario action matrix, which is the tabular depiction of the UCDM. Importance can also be represented through a coloring or shading of the rows, such as red for most important and progressing through the color spectrum through blue for least important.

The ultimate goal of use case dialog maps is to assist in the feature analysis management. We should be able to take a normalized use case dialog map to assess the effort necessary to implements certain sets of functionality. From this point we should be able to analyze marginal rates of efforts to provide additional functionality. Since our concern is in a domain specific environment, the leveraging of reusability and replacability of functionality in a framework can be assessed, as well as the cost or impact of certain design decisions. In other words, what is the cost of providing this level of flexibility and how much will this designed variability cost us over the life of the domain architecture? Estimates can be made to assist in the decision, but then we should be guided in assessing the actual ramifications of that decision.

7.3.2. Sequence Diagrams

The UML identifies sequence diagrams are one of several notation devices for representing object-oriented models. These diagrams show the sequence of messages that are passed between model elements. The presentation of sequence diagrams to illustrate use cases can be puzzling, because it reveals an inconsistency in what is promoted. As discussed in chapter five, use cases should not be solely presented from the perspective of the actor, since in many cases these would be rather trivial – often consisting of a single event and response. Those scenarios can best be described through contracts or responsibilities. Where use cases really become of value is when they take a control centric perspective – as opposed to actor (interface) or data centric. For example, Figure 7.7 illustrates a simple recycling machine message event trace. The user, or actor, simply deposits items and asks for a receipt. The really interesting activity centers on the deposit item receiver. This entity is involved in determining the type of deposit item and the value to be credited. It also coordinates the display of the amount credited, cash dispensing, and receipt printing. By considering object roles of the collaborating entities, the focus of interest can be revealed. Several classifications may be applied to the sequence diagram, such as the Objectory process stereotypes [JCJO92], the Wirfs-Brock stereotypes [WB93], or the Chafi behavioral life-cycle stereotypes [Chaf96]. In Figure 7.7, Chafi's event, recognition, communication (ERC), decision (D), and transaction (T) lifecycle roles have been applied. They reveal that the primary focus of interest is the Deposit Item Receiver, since virtually all of the interactions involve this entity.

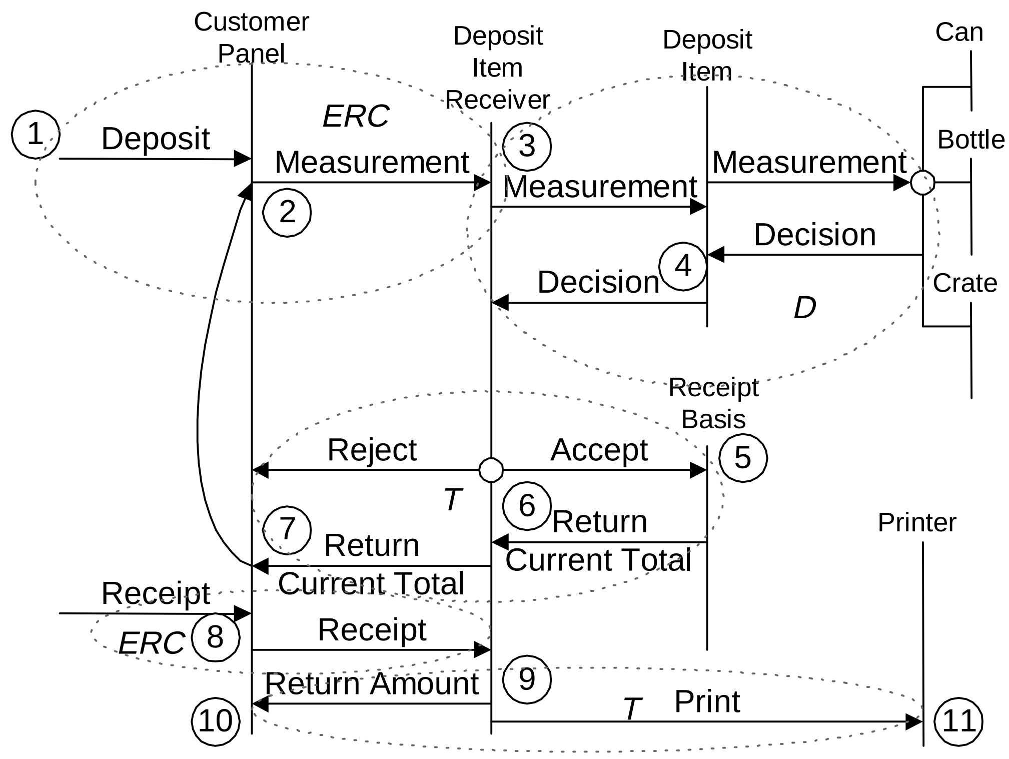
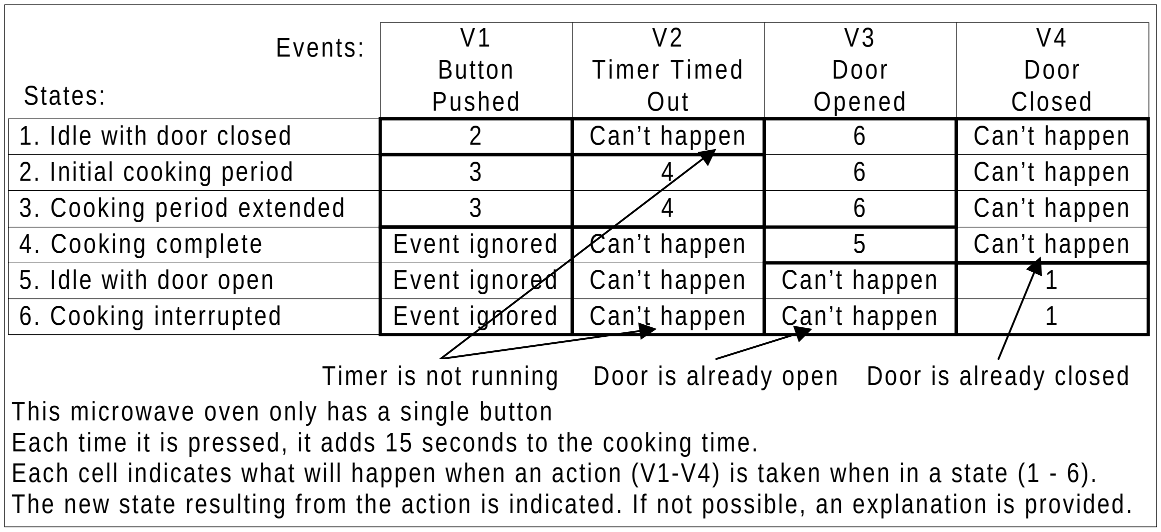


Figure 7.7. Recycling Machine Sequence Diagram

7.3.3. Event-State Matrices

An event-state matrix can be automatically transformed from the event flow diagram. State machine elicitation techniques [ML97] can be extended to incorporate additional types of analysis stereotypes, rather than being constrained to only the Objectory interface, entity, and control object types. A key event dictionary [WF97] may also be used to assist in defining the event-state matrix. Both of these techniques were describe in the chapter three. This event-state matrix in turn is used to generate a set of case statements that abstracts, or factors out commonality. This is accomplished through a semi-automated process that examines clusters of related states and events in the table to define the set of unique cases. Specifically, if an event triggers a state transition from two or more states to the same target state, then these source states may be clustered. This process will identify where changes occur and create "zones" of same states or same transitions. For example, Figure 7.8 illustrates a simple microwave oven that identifies the clustering of state zones. For some states, it may only be possible to reach that state by progressing from one or more other states. In this example state V4 can only be reached via state V3. Thus, this also represents a potential clustering. The cluster analysis algorithm performs a "problem decomposition" that minimizes the shared functions from the event-state matrix clusters.



Events: / States:	V1 Button Pushed	V2 Timer Timed Out	V3 Door Opened	V4 Door Closed
1. Idle with door closed	2	Can't happen	6	Can't happen
2. Initial cooking period	3	4	6	Can't happen
3. Cooking period extended	3	4	6	Can't happen
4. Cooking complete	Event ignored	Can't happen	5	Can't happen
5. Idle with door open	Event ignored	Can't happen	Can't happen	1
6. Cooking interrupted	Event ignored	Can't happen	Can't happen	1

This microwave oven only has a single button
Each time it is pressed, it adds 15 seconds to the cooking time.
Each cell indicates what will happen when an action (V1-V4) is taken when in a state (1 - 6).
The new state resulting from the action is indicated. If not possible, an explanation is provided.

Figure 7.8. Microwave Oven: Event-State Matrix Cluster Analysis

Some event-state matrixes are very sparse. The focus for these types of matrices has typically been on hits, i.e. cells where an anticipated action will take place. The cluster analysis technique presented here systematically address all cases. It is also desirable to identify those that can be ignored and those that can't or shouldn't happen. Often this relates to a risk analysis. For example, in an employee personnel administrative system, an employee may go through certain employment events and states, such as hired (active), military alert (leave of absence), death (terminated), and rehired (active). Certain event sequences are expected not to happen, but they might occur. This is where feature selection and risk assessment come into consideration. Thus, cluster analysis also provides a "framework for discussion" by serving as a mechanism to quantify the dialog between development and the client. For example, a developer may point out two places revealed by the cluster analysis when potential risk is present, i.e. in cells marked as "can't happen". The client must then consider the cost and whether or not he is willing to pay for them.

Restraints are placed on full automation of the cluster analysis process in order to maintain integrity of the conceptual domain model. Although some clusters may be logically correct, they may not be practical from the perspective of long term evolution of the domain model. A concept map of the domain architecture is used as a device to assist in making this assessment.

7.3.4. Code Segmentation

Code segmentation is represented through a collection of event-condition-action statements that consist of three scenarios. In the first scenario, the state and related set of conditions to arrive at the state will dictate certain actions that will be taken. Each event-condition-action statement constitutes a rule that can document design. Moreover, these statements match the structure of the business rule pattern presented in chapter six. Thus, they serve as a conduit to introduce parameterization and apply normalization from the policy view of the domain model. In the second scenario, certain event-states combinations will be defined as "can't happen". Typically, a significant portion of event-states will fit this profile. These "can't happen" scenarios are of interest to the developer in order to generate failure code. The third scenario is that the event is ignored, a NO-OP. The developer should be able to implement from these event-condition-action case statements with full confidence that system behavior will be verifiable. Transformation into the event-condition-action case scenarios provides a novel approach to testing and validation. The microwave oven example is continued in Figure 7.9. The "STATE 1" case statement identifies the event and either of two states that the oven can be in to reach this state.

```
CASE
  STATE 1:                   v4(state 5 ∪ state 6)
  STATE 2:                   v1(state 1)
     •
     •
     •
  STATE 6:                   v3(state 2 ∪ state 3)
  CAN'T HAPPEN:              Error Code
  EVENT IGNORE:              NO-OP
END
```

Figure 7.9. Microwave Oven Case Statement Automatically Generated from Event-State Matrix

When dealing with the case scenarios, we need to consider the number of scenarios from the event-state matrix to address performance issues. Additionally, the anticipated frequency of each scenario must be taken into account. Overall requirements may dictate certain performance levels (e.g. response in three seconds is often cited for GUI response). Dynamic structuring may be specified through business rule patterns.

7.3.5. Business Rules

The relationship between parameterized business rules and adaptive use cases have been described in the two previous chapters. The event-condition-action statements from the code segmentation representation already provide normalized rule format. The developer only has to apply one or more parameterized business rule patterns to the expression to leverage reuse of domain architecture artifacts.

7.3.6. Regular Expressions

The final representation in the DEAT process model is transformation of the event-states into regular expressions. These event sequence expressions are of particular interest to the designer since they can be transformed directly back into use cases. The syntax for the regular expression language use here is adopted from the Beringer's object life-cycle grammar [Beri97]. The following operators are used: Sequence (A.B), interleaving or concurrent services (A || B), optional service ([A]), alternation (A | B), repetition of zero or more (A*), repetition of one or more (A+), interleaving or concurrent repetition (A ||*, A||+), and precedence ([], *, +, ||*, ||+, ., |, ||).

Regular expressions provide validation of design. For example, in Figure 7.10, we once again continue our example of the microwave oven. The first expression simply represents an opening and closing of the door. This is the use case for placing the food in the oven, but it could also be a use case for someone cleaning the oven. This latter one may have been overlooked in the original use case scenarios. However, on the basis of discovering the new cleaning scenario, which has a different actor (a cleaner instead of a cook), we could look at the event V3 and determine if we want the light to turn on when the door is open for each of these scenarios. The second expression represents the main use case for the oven. This expression shows the button pushed one or more times, then the cooking period completing. The door is opened to retrieve the meal and then finally closed. The last expression in the figure is also part of the main use case, however, this is an alternative scenario since we normally expect the cooking cycle to complete. This scenario raises additional design issues, such as: do we delay the opening of the door to make sure that radiation doesn't leak? Also, do we reset the timer when the door is closed or do we resume? If we reconsider the second expression representing the main path of this use case, then we might reconsider what other actions could end the cooking cycle other than the timer, such as loss of power or a thermal sensor tripping. Consideration of these alternative path scenarios might not seem relevant, but they provide the opportunity to consider future enhancements, perform a risk assessment as part of the "framework for discussion" described earlier, or shift the focus from "good design" to "legal design", such as meeting safety standards in this example.

V3.V4
Or
$V1^{+}$.V2.V3.V4
Or
$V1^{+}$.V3.V4

Figure 7.10. Microwave Oven Event Sequence Regular Expression

One other advantage of the transformation to regular expressions is to identify "anti-scenarios" – those sequences of states that shouldn't happen as opposed to those that can't happen. Ultimately we are looking at the regular expressions for defining what is a good dialog, based on a "good lifecycle".

Object roles provide one way to validate a "good lifecycle". Computer/human interaction should also be considered when establishing good "dialogs". Defining what constitutes a "good lifecycle" is important in evaluating a system that behaves effectively. This involves standardization in behavior domains that are specific to the particular problem domain under consideration.

Several approaches can fit smoothly into this structure of regular expression – e.g. the Flores/Winograd speech acts model, Chafi's behavior lifecycle, and Jacobsen's objects types. What may be of even greater significance is the ability to migrate from one set of capabilities to another by first mapping them into the framework in order to provide a two step transformation, similar to many image conversion processes. For example, although the Chafi set can map into the Jacobsen set directly, it becomes more effective to be able to select the most appropriate technique depending on the stage of the domain model evolution.

These transformations fall out automatically. However, there are some problems that must be addressed when information is abstracted and may be lost by using a certain approach. By extending certain sets of capabilities to conform to the framework, we will be able to provide lossless transformations.

In many respects, the regular expression can be compared to propagation pattern constraints in the adaptive software approach. Where adaptive software defines an object path that messages or return data must traverse, event sequence regular expressions can stipulate states that must be traversed to arrive at valid cases.

The leveraging of regular expression can be best appreciated when considering a family of products within a problem domain. Initially we may be dealing with a single product. The regular expression allows us to identify where enhancements will be made for the next product release. Later, when considering a family of products, it assures a consistent set of behaviors. Consider, by analogy, the dashboard layout in a family of automobiles. An auto manufacturer, such as General Motors, maintains relatively consistent functionality and behavior across all models and even across product lines.

As demonstrated in the discussion of Figure 7.10, once we determine where a modification or enhancement should be made through analysis of the regular expression, we are able to isolate existing functions. We are then able to choose whether to eliminate the function or enhance it. If we choose to enhance, it may be in the form of splitting up the behavior that previously was shared across a set of event-states in the event-state matrix. Of course, we can also define a new event, which entails either adding/extending states or adding more events. Moreover, regular expressions also invoke discussion among architects and domain experts to uncover errors, missed assumptions, and other anomalies.

7.4. Cluster Analysis and Transformation Example

Case Study Premise. An organization offers three related dealer training course. Each is conducted around ten times per year. The basic course is required for all new dealers and territory representatives. Existing dealers often send recently hired managers. New dealers must also attend a separate follow-up course approximately six months after opening or be in violation of their contract. A refresher course is also offered for existing dealers. This refresher course has two sections. The first for business-to-business dealers, the other for retail stores. These sections are held in conjunction with the basic course. Each basic course attendee attends either one or the other section as part of their enrollment. If the course is filled, then existing dealers may get bumped into the next scheduled class. New dealers (along with their additional attendees, if any) and new territory managers are never bumped.

Event-State Matrix Analysis. We start out by identifying the core events and states associated with a state in Figure 7.11. This may be the result of finding a compatible component with its related documentation. Nonetheless, analysis reveals that all three courses can fit into the same event-state matrix.

	e1 Prior Class Closes	e2 Enrollment Begins	e3 Student Enrolls	e4 Student Drops	e5 Enrollment Closes	e6 Course Begins	e7 Course Ends
s1 Scheduled	SH	s2	×	×	×	×	×
s2 Open For Enrollment	[Refine]	×	[Refine]	[Refine]	s3	×	×
s3 Closed For Enrollment	×	×	×	×	×	s4	×
s4 In Session	×	×	×	×	×	×	s5
s5 Completed	×	×	×	×	×	×	×

Figure 7.11. Dealer School High Level Event-State Matrix

This first event-state matrix reveals a cascading event pattern. Although the exercise of examining this may seem trivial, it demonstrates a “stable” pattern that we will want to maintain. The choice of the time aspect as the focus of abstraction at this level hides the details of the booking states for the overall course and the breakout sessions. Since these aspects involve complex business rules, they would only tend to clutter up this diagram and hide the overall simplicity of this pattern. The advantage of this approach is identification of components that can be reused or purchased that fill a basic need. The cells identified are “Refine” are places where plug-in component modules can offer tailored customization. One other area of interest is the “Shouldn’t happen” (SH) cell. This has most to do with frequency of offering the course. It identifies a situation where enrollments would not be able to be taken. A component selection process will have to take this feature into consideration. This also reveals a workflow issue that requires scheduling of the following year’s classes prior to the closing of the last class in the current year. The chunking principle [Horn89] of 7±2 needs some modification for matrices. Alternative complexity weightings that can be used are number of cells, axes, and cell contents. The Figure 7.11 matrix has 35 cells, 12 axes values (five states and seven events), and eight cells of interest (i.e. those labeled other than “can’t happen” or “no-op”).

The event-state matrix for the cells marked as “Refine” in Figure 7.11 is constructed and shown in Figure 7.12. This matrix appears far more complex although it is much smaller: 20 cells, nine axes values, and 15 cells of interest. What appeared to be a very simple state transition matrix at a higher level of abstraction reveals significant complexity at a more detailed level. Here, we selected the events identified in the prior matrix and expanded the types of events and states into sub-states and typed events. The bolded cells identify transitions to multiple states. This situation reveals an ill defined pattern.

		e1a Bumped Student Rescheduled	e3a Priority Student Enrolls	E3b Non-Priority Student Enrolls	e4a Priority Student Drops	e4b Non-Priority Student Drops
s2a	Unfilled	s2b	s2b	s2b	×	×
s2b	Partially Filled	**s2b, s2c**	**s2b, s2c**	**s2b, s2c**	**s2a, s2b**	**s2a, s2b**
s2c	Filled	**s2c, s2d**	**s2c, s2d**	**s2c, s2d**	**s2b, s2c**	**s2b, s2c**
s2d	Overbooked	s2d	s2d	s2d	**s2c, s2d**	**s2c, s2d**

Figure 7.12. Dealer School Decomposition Event-State Matrix

We have three choices at this point: further decomposition, transformation to a different representation, or applying a minimal clustering algorithm to simplify the existing diagram. We will look at each to see what can be revealed. First, by applying minimal clustering (Figure 7.13), we consolidate the common cells to reveal only seven cells that are actually of interest, of which only five are unique.

		e1a Bumped Student Rescheduled	e3a Priority Student Enrolls	e3b Non-Priority Student Enrolls	e4a Priority Student Drops	e4b Non-Priority Student Drops
s2a	Unfilled	A = s2b			×	
s2b	Partially Filled	**B = s2c, s2b**			C = s2b, s2a	
s2c	Filled	**D = s2c, s2d**			**B**	
s2d	Overbooked	E = s2d			**D**	

Figure 7.13. Dealer School Clustered Decomposition Event-State Matrix

This normalized representation looks suitable for transformation to another representation, namely business rules. The heuristic used here is to look for something that is causing a decision to be made as to whether to remain in the same state or change states. We could try further decomposition of the boundary states, such as when only one student is enrolled or there is room for only one more student, as shown in Figure 7.14.

		e1a Enroll	e3a Drop
s2a	Unfilled	s2b1	×
s2b1	One Student Enrolled	**s2b2**	s2a
s2b2	Partially Filled	s2b2,s2b3	s2b1, s2b2
s2b3	One Space Left	s2c	**s2b2**
s2c	Filled	s2c, s2d	s2b3, s2c
s2d	Overbooked	s2d	s2c, s2d

Figure 7.14. Dealer School Clustered Decomposition Event-State Matrix II

With the exception of exploring the boundary conditions, nothing new is revealed by this decomposition. A heuristic that may be applied is that if the complexity increases through decomposition, then attempt to normalize. If the complexity remains with no significant change in the nature of the presentation, then try a different transformation. “Nature of presentation” refers to the pattern that emerges from the matrix. An “event cascade pattern” was recognized in the first matrix. This was replaced by a dense pattern in the second matrix. Having made no further progress with event-state matrix decomposition or normalization, a transformation should be applied.

Business Rule Transformation. The first thing that becomes evident when transforming the event-state matrix into a code segmentation representation of events, conditions, and actions is the need to define the events and states as formal expressions, as shown in Table 7.6. In the process, the need to define a parameter becomes apparent: *max_enrollment*.

Table 7.6. Formal Expressions for Code Segmentation Conditions

Key	Description	Expression
e1a	Student Rescheduled	course_enrollment = course_enrollment + 1
e3a	Priority Student Enrolls	course_enrollment = course_enrollment + 1
e3b	Non-Priority Student Enrolls	course_enrollment = course_enrollment + 1
e4a	Priority Student Drops	course_enrollment = course_enrollment - 1
e4b	Non-Priority Student Drops	course_enrollment = course_enrollment - 1
s2a	Unfilled	course_enrollment = 0
s2b	Partially Filled	course_enrollment > 0 and course_enrollment < max_enrollment
s2c	Filled	course_enrollment = max_enrollment
s2d	Overbooked	course_enrollment > max_enrollment

On the surface, mapping the expressions from Figure 7.14 looks like straightforward, simple math. However, the filled state (s2c) reveals an interesting anomaly. Specifically, how can adding a student when filled result in a partially filled class? Yet this situation arises due to the dynamics of priority students and bread-out sessions. For example, if the retail store section is filled and a new dealer enrolls (with a business-to-business breakout session), then an existing dealer student currently enrolled in retail store will get bumped. This frees up space in the course enrollment that was limited by a breakout session constraint rather than the overall course enrollment constraint. This transformation process ultimately reveals that three factors need to be considered when a class is overbooked:

- Type of student (i.e. priority or non-priority). This can best be addressed by examining the student entity.
- The breakout session selected. In isolation, this is only significant if the size of the breakouts is less than the total size. For example if maximum enrollment is 100, but the maximum size of either

breakout session in 70. Any unconstrained breakout enrollment would set each breakout session size to the maximum enrollment for the course.

- The influence of enrollment in the refresher course. An unconstrained influence would bump all students enrolled in the refresher course first. An example of a constrained influence would be to guarantee twenty percent of all section seating for refresher course enrollees.

We also consider the impact of when a student gets bumped – either immediately after an enrollment puts the class in an overbooked state, or at the time that the class is closed. This can be influenced by the architectural decision of batch versus on-line. Figure 7.15 illustrates the event-state matrix for an on-line architectural with immediate bumping would look like. Notice that the student type has been removed, since it provided no distinction in state. This is also a result of minimization clustering to normalize the event-state matrix. The only significant change between this matrix and a matrix for a batch architecture is the "student bumped" event while overbooked. It will always result in a restoration of the filled status only, since all students that can get bumped will get immediately bumped.

		e1a Bumped Student Rescheduled	e3a Student Enrolls	E4a Student Drops	e4b Student Bumped
s2a	Unfilled	s2b		×	
s2b	Partially Filled	**s2b, s2c**		s2a, s2b	
s2c	Filled	**s2c, s2d**		**s2b, s2c**	
s2d	Overbooked	s2d		**s2c, s2d**	s2c

Figure 7.15. Dealer School Normalized Event-State Matrix for an On-line Registration System

We can also use this event-state matrix for the section states. Consideration of sections leads us to another consideration: if a section is closed, the course is not necessarily closed. Therefore, we can reapply this pattern to deal with the states are referring to sections filled rather than student enrollments.

		e1a Bumped Student Rescheduled	e3a Student Enrolls	e4a Student Drops	e4b Student Bumped
s1	No sections filled	**s1, s2**		s1	×
s2	Some sections filled	**s2,s3**		**s1,s2**	
s3	All sections filled	s3		**s2,s3**	

Figure 7.16. Dealer School Normalized Event-State Matrix for Section States

We can now fully apply a transformation of the event-state matrix to business rules. We start with the section states just defined in Figure 7.16. These are shown in Example 7.1.

State:	No sections filled (s1)
Expression:	$\forall$ s:S $\cdot$ s.(se < mse)
English equivalent:	For all sections, the current enrollment is less than the maximum enrollment
State:	Some sections filled (s2)
Expression:	$\exists$ s:S $\cdot$ s.(se $\geq$ mse)
English equivalent:	There exists at least one section in which the current enrollment equals or exceeds the maximum enrollment
State:	All sections filled (s3)
Expression:	$\forall$ s:S $\cdot$ s.(se $\geq$ mse)
English equivalent:	For all sections, the current enrollment equals or exceeds the maximum enrollment

Where:	S = section
	C = course
	se = section enrollment
	mse = maximum enrollment for that section
	ce = course enrollment
	mce = maximum enrollment for course
	s $\in$ c
	S(se,mse) $\in$ C(ce,mce)

Expanded version of expression:	$\forall$ s:S(se,mse) $\in$ c:C(ce,mce) $\cdot$ s.(se < mse)

Example 7.1. Section State Business Rule Formal Expressions

Having dealt with section substates, event types become the next focus of attention. We begin first with a definition of priority and non-priority students. It should be noted that the transformation and manipulation of these representations is presumed to be performed by a domain engineering support tool. Such a tool would utilize a visual programming language, with the description and English equivalent as the key devices that are manipulated. The logical expression is maintained for internal representation by a domain support tool, but can be viewed and manipulated if desired. Through a combination of menu selection and dialog, an expression is build. The English equivalent is a by-product that uses an algorithm to condense the expression verbiage. Example 7.2 shows the event type business rule formal expressions for priority and non-priority students and the bumping event.

Entity:	Priority Student (ps)
Expression:	$ps \in N{:}T \wedge ps \in B{:}T$
English equivalent:	A priority student is a member of the group of new students or bumped students
Entity:	Non-Priority Student (nps)
Expression:	$nps \notin N{:}T \vee nps \notin B{:}T$
English equivalent:	A non-priority student is not a member of either the group of new students or bumped students
Entity:	Most Recently Closed Class for a Course (mrcc)
Expression:	$c{:}C(cd) \cdot (cd < = today() \vee c.(cd \equiv MAX(CC.cd))))$
English equivalent:	The Most Recently Closed Class for a Course is the class that has a closed date that is less than or equal to today's date and is greater than (more recent) that any other closed class for that course.
Expression:	$count(NP(c,s,ed) \cdot ((bs(s) = NP(s) \vee bs(c) = NP(c) \vee bs(ed) > NP(ed))$
English equivalent:	number of non-priority students that enrolled after the bumped student
Entity:	Bumped Student (bs)
Expression:	$bs \in NP{:}T(c,s,ed)) \cdot (\forall nps{:}NP(c,s,ed) \cdot (s{:}S \in c{:}C(mrcc) \vee s.(se - mse) > count(NP(c,s,ed) \cdot ((bs(s) = NP(s) \vee bs(c) = NP(c) \vee bs(ed) > NP(ed)) \wedge (c.(ce - mce) count(NP(c,s,ed) \cdot bs(c) = NP(c) \vee bs(ed) > NP(ed))))$
English equivalent:	A bumped student is a non-priority student that belonged to a section of the most recently closed course of a given type and the number of overbooked students in that section exceeds the number of non-priority students that enrolled after the bumped student or number of overbooked students in that course exceeds the number of non-priority students that enrolled after the bumped student.
Precondition:	Bumping Event (Be)
Expression:	$\exists nps{:}T \cdot nps.(nps \in s{:}S \cdot s.(s.se \geq s.mse))$
English equivalent:	There exists at least one non-priority student enrolled in a section that is filled or overbooked.

Where:	ps = priority student
	nps = non-priority student
	T = student type
	N = the set of all new students
	B = the set of all students bumped from the prior class
	S = section type
	s = section instance
	se = section enrollment
	mse = maximum enrollment for that section

Expanded Bumped Student expression:	$bs \in NP{:}T(c,s,ed)) \cdot (\forall nps{:}NP(c,s,ed) \cdot (s{:}S \in c{:}C(cd) \cdot (cd < = today() \vee c.(cd \equiv MAX(CC.cd)))) \vee s.(se - mse) > count(NP(c,s,ed) \cdot ((bs(s) = NP(s) \vee bs(c) = NP(c) \vee bs(ed) > NP(ed)) \wedge (c.(ce - mce) > count(NP(c,s,ed) \cdot bs(c) = NP(c) \vee bs(ed) > P(ed))))$

Example 7.2. Event Type Business Rule Formal Expressions

After doing this exercise, it becomes obvious that we can reconsider the structure of the course offering to include two sections (such as a lecture and lab). If either one fills up, you are restricted from taking the other. This adds an additional level of complexity that the client must choose to provide and deal with, or sacrifice. For example, the client may choose the simpler implementation and decide on manual intervention flags to handle exceptional cases if they arise. Thus, the following provisions are put in place:

- Section size will equal course size.
- Refresher course students will always be admitted.
- A manual review of the class roster will be make prior to final confirmations and/or bumping if there are any over-bookings.

7.5. Discussion

Perhaps the best way to characterize the DEAT process model and methodology is to declare what it is not. It is not a generative development methodology, but does work in adjunct with such development methodologies by leveraging corporate assets in domain model artifacts when developing new applications. It is also not a static collection of techniques and representations. Although specific use case formalisms are described in the Figure 7.1 process model, these are only a representative set of a large number of tools that may have been included. Chapter three enumerated a significant number of additional representations that may be suitable for many domain model problem areas.

Recent trends show that commercially available software tools and techniques are driving the design process and defining the methodologies. Methodologies have become fluid as software engineering tools frequently change to match features of competitors. The individual needs of domain-specific architectures only add to this trend. With the widespread acceptance of the UML specification, a base set of models provides the common denominator for domain-specific models. The ability to map specialized representations into UML modeling constructs encourages diversity that can be tailored not only to specific domains, but also to specific personality types.

The extended example in the previous section attempted to focus on the core purpose of the DEAT methodology – the utilization of a family of use case formalisms that reveal hidden nuances of domain architecture evolution. This discovery process allows us to maintain a domain architecture of non-interfering applications and minimize bias towards any individual application. Hence, as the domain model is expanded to accommodate new applications, future applications will not be unnecessarily constrained or complex.

The DEAT methodology is not so much concerned with individual techniques as it is with the patterns of relationships between them. Techniques, which seem like elementary building blocks, keep varying from one domain-specific environment to another – sometimes slightly, sometimes significantly. When the nature of these techniques is examined through the stable perspective of the domain engineering conceptual framework from chapter four, the techniques themselves dissolve and only the fabric of the relationships among them remain. In other words, one representation may reveal a recognized pattern that guides the discovery process, such as the transformation from the event-state matrix in Figure 7.14 to the formal expressions for code segmentation conditions shown in Table 7.6.

As more tools and techniques are added to the domain architect's arsenal, it is expected that a well-defined collection of heuristics will emerge. This speculation is based on two assumptions. First, standardization of domain model constructs will continue to encourage the incorporation of new representations and techniques into modeling tools. Major product offerings will incorporate these techniques as plug-in modules through an extensibility mechanism or provide for the import and export of model fragments with specialized stand-alone applets that support a new technique. The second assumption is that experience with representations in several domains will provide a knowledge base of useful transformations tied to the characteristics of the source representation that led to the transformation. An Internet-based common repository that implements collaborative filtering is envisioned. Through such a facility, similarities in problem domain, personality type, and stakeholder can be exploited.

The most structured portion of the DEAT process model is the deployment evaluation sub-model. These component representations are intended to bridge the cognitive gap between management and software engineers. The configuration model extends the concept of parameterization by demonstrating how the deployment of an application can be facilitated through application of the business rule pattern language developed in chapter six. The fit assessment and change costing analysis component models quantify each of the domain normalization operations through metrics that can calculated and then used to gauge the maturity of the domain architecture through its defined attributes. This provides management with valuable information that begins to migrate the complex and ill-defined problem of managing domain-specific architecture evolution to an environment that emulates the features associated with a disciplined engineering process rather than an artistic one.

CHAPTER VIII

CONCLUSION

8.1. Contributions

This thesis described two primary concerns when developing an application from a domain architecture. The first concern was how to maintain non-interfering applications derived from the domain architecture. The second concern was how to minimize bias towards any individual application in developing the domain model. The domain evolution architectural transformation (DEAT) process model was presented to address these concerns. It enables users to bridge the cognitive gap that often exists between management and software engineers. It also fosters the application of domain knowledge, design principles, standards, and programming skills for generating an application and evolving the domain architecture in such a way that desirable properties are maintained. Several contributions are made through this research:

1. The domain engineering conceptual framework establishes a context for the development of a domain architecture and deployment of applications from that architecture.
2. The principles of domain model normalization provide the basis for performing operations on the domain architecture through four mechanisms: use case formalisms, componentization, parameterization, and transformation.
3. Domain model normal forms describe how normalization of four views on a domain model interact to provide various levels of normal form for structural, process, rule, and context sub-models.
4. A use case refinement level model enables view focus to be tailored to various stakeholders. This model provides a context for viewing higher level use case actions by lower level stakeholders, such as designers. Thus, it provides more flexibility that any other use case modeling approach.
5. The adaptive use case model provides a comprehensive extension to the Unified Modeling Language (UML) specification, including detailed well-formedness rules expressed though the Object Constraint Language (OCL).
6. A business rule meta-model integrates workflow activities with use case formalisms that provide a basis for formalizing business rules as part of the domain architecture specification.

7. A comprehensive parameterized business rule pattern language permits system requirements to be expressed in terms of a reusable configuration language through parameter value bindings.
8. The domain evolution architectural transformation (DEAT) process model provides the basis for a methodology that is specifically focused on the maintenance aspects of a domain architecture through a series of sub-models that help define appropriate domain normalization operations.
9. The configuration model and its application logic and presentation logic sub-models prescribe a formal structure to support parameterized configuration environments for domain architectures.
10. The fit assessment model prescribes a quantitative measure of changes to the domain architecture that are needed to support new system development activities. Process, selection, presentation, persistence, and interfaces complexity levels are used to develop a set of metrics.
11. The exception analysis and change costing models provide a vehicle to tie the technical aspects of domain evolution to management concerns with respect to personnel, resources, and costs.
12. Use case dialog maps provide a new graphical representation of use cases that facilitate iterative transformations as prescribed in the DEAT process model.
13. A use case formalism sub-model prescribes how a series of use case transformations may be manipulated in order to reveal nuances about changes to the domain architecture.

Although considerable work remains, this thesis demonstrates the feasibility of utilizing the prescribed models for guiding performance of operations on a domain model. Through fit assessment and change cost analysis, new applications may be developed from the domain architecture with minimal bias and interference resulting in a more stabile and resilient domain model.

8.2. Future Work

This research has focused primarily on providing the conceptual basis for performing domain normalization operations to facilitate the evolution of a domain architecture. As suggested in the introduction to this thesis, it is likely that a long-term research agenda will be required to develop a complete set of formalisms to fully develop each facet described in the conceptual framework. This effort is only taking the first incremental steps in defining such a framework. Extensive empirical evidence in a wide range of domains will be needed to validate the usability of new derived formalisms.

Many of these models have been applied in actual production development environments, so some empirical knowledge has already been acquired. Four business domains have utilized some or all of the models presented in this thesis. These include development of domain architectures to support system development for multi-tier dealer training, benefits administration, retail advertising layout, and forms management.

Future plans include work in the following areas:

1. Development of a visual programming environment to support elaboration of rules derived from the business rule patterns. This environment would also provide automated assistance in developing a multi-dimensional domain specific language.
2. Facilities for discriminating between shared business rules across applications and rules that are merely copied but no longer bound to the original rules.
3. Automation tools to facilitate the configuration parameters that specify build-time versus run-time bindings in order to leverage flexibility/performance tradeoffs.
4. Expansion of the business rule pattern library through more domain specific constructs. The objective is to discover fundamental business rule meta-patterns that may be applied to any domain architecture in order to provide greater domain architecture stability.
5. Development of a more formal domain architecture maturity model based of the domain architecture attributes and operations.
6. Development of a set of algorithms to summarize the narrative text representations of *NarrativeScenarios* into a *NarrativeUseCase.*
7. Definition of domain architecture documentation sets specifically tailored to the maturity level of a domain architecture and the personality type of the user. Dynamic generation of documentation would integrate a collaborative filtering mechanism.
8. Formalizing a query language for domain architecture artifacts that leverages the normalized properties of a domain model. Such a language would provide the linkage between the domain artifact repository and the visual programming environment.
9. Formal elaboration of composition rules for a domain model that preserve normal form properties.

BIBLIOGRAPHY

[AAKL93] Auer, A., Alanko, J., Karjalainen, J., Lintulampi, R., "User need oriented software component development", IEE Colloquium on 'Software Instrumentation – Software Components (Digest No.011), pp. 2/1-3, 1993.

[AB95] Andersson, M., and Bergstrand, J., Formalizing Use Cases with Message Sequence Charts, M.S. Thesis, Department of Communication Systems at Lund Institute of Technology, 1995. (http://www.efd.lth.se/~d87man/EXJOBB/PS/ExBook.ps.Z.uu on 1/4/98)

[ABS95] Armour, F., Boyd, L., and Sood, M., "Use case modeling concepts for large business system development", OOPSLA workshop – Requirements Engineering: Use Cases and More, 10/15/95, Austin, Texas, 1995.

[ABV92] Aksit, M., Bergmans, L., and Vural, S., "An Object-Oriented Language-Database Integration Model: The Composition-Filters Approach", ECOOP '92 Proceedings, 6/29/92, Utrecht, The Netherlands, pp. 373-395, 1992.

[AC94] Adams, G., and Corriveau, J., "Capturing Object Interactions", Proceedings of TOOLS Europe '94, Versailles, 1994.

[Adai95] Adair, D., "Building Object-Oriented Frameworks - Parts 1 and 2", AIXpert, Feb. 1995 and May 1995. (http://www.developer.ibm.com/sdp/library/aixpert/feb95/aixpert_feb95_boof.html and http://www.austin.ibm.com/developer/aix/library/aixpert/may95 on 5/24/96)

[ADP95] Alvarez, X., Dombiak, G., and Prieto, M., "Use-cases, interaction diagrams, hypermedia and visualization", OOPSLA workshop – Requirements Engineering: Use Cases and More, 10/1595, Austin, Texas, 1995. (http://www.unantes.univ-nantes.fr/usecase/Contributions/Lifia.UseCase.Final.ps on 2/5/97)

[AG90] Anderson, D. B., and Gossain, S., "Software reusability using object-oriented programming", IEE IT Conference, 3/19/90-3/22/90, Southampton, UK, pp. 299-305, 1990.

[AG96] Appley, G. and Gallaher, M., "A Framework for Manufacturing-Process Software", Object Magazine, 6(3), pp. 33-41, May 1996.

[Alex79] Alexander, C., The Timeless Way of Building, Oxford University Press, New York, 1979.

[Alex79a] Alexander, C., A Pattern Language, Oxford University Press, New York, 1977.

[Ambl97] Ambler, S., "Normalizing Classes", Software Development, 5(4), May 1997.

[Ande94] Andert, G., "Object frameworks in the Taligent OS", Spring COMPCON 94 Digest of Papers, 2/28/94, San Francisco, CA, (Cat. No.94CH3414-0), pp. 112-21, IEEE Computer Society Press, 1994.

[AP92] Alagar, V., and Periyasamy, K., "A methodology for deriving an object-oriented design from functional specifications", Software Engineering Journal, 7(4), pp. 247-63, Jul. 1992.

[AR94] Al-Yasiri, A., and Ramachandran, M., “Developing software systems with domain oriented reuse”, EUROMICRO 94 Proceedings: System Architecture and Integration, 9/5/94, Liverpool, UK, ISBN: 0 8186 6430 4, pp. 133-9, 1994.

[ASP93] Arango, G., Schoen, E., Pettengill, R., “A Process for Consolidating and Reusing Design Knowledge”, IEEE, Cat 0270-5257/93, pp. 231-242, 1993.

[Auge91] Augeraud, M., La Rochelle, I., and Freeman-Benson, B., “Dynamic Objects”, 1991 ACM SIGOIS Conference on Organizational Computing Systems, SIGOIS Bulletin 12(2,3), pp. 129-134, 1991.

[BaH89] Barnes, P., and Hartrum, T, “A decision-based methodology for object-oriented design”, Proceedings of the IEEE 1989 National Aerospace and Electronics Conference NAECON 1989 (Cat. No.89CH2759-9), New York, NY, Vol.2, pp. 534-41, 1989.

[Bass96] Basset, P., “Framing Software Reuse: Lessons From the Real World”, Prentice Hall PTR, Upper Saddle River, New Jersey, ISBN 0-13-327859-X, 1996.

[BB91] Barns, B., and Bollinger, T., “Making reuse cost-effective”, IEEE Software, 8(1), pp. 13-24 Jan. 1991.

[BB95] Berteaud R., and Bézivin, J., “Requirement modeling in the OSMOSIS workbench”, OOPSLA – First Workshop on Use Cases, 10/15/95, Austin, Texas, 1995. (http://www.unantes.univ-nantes.fr/usecase/Contributions/berteaud.ps on 1/11/98)

[BC92] Buhr, R., Casselman, R., “Architectures With Pictures”, OOPSLA '92, pp.466-483, 1992.

[BDRV91] Beck, R.; Desai, S., Radigan, R., and Vroom, D., “Software reuse: a competitive advantage”, IEEE International Conference on Communications Conference Record, 6/23/91, Denver, CO, Vol.3, pp. 1505-9, 1991.

[BE93] Birrer A., and Eggenschwiler, T., “Frameworks in the Financial Engineering Domain: An Experience Report”, ECOOP '93 – Object-Oriented Programming 7th European Conference, Kaiserland, 7/26/93, Germany, pp. 21-35, 1993. (http://www.ubilab.ubs.ch/paper/Bir93b.ps.Z on 5/3/96)

[Beed97] Beedle, M., “A ‘light’ distributed OO Workflow Management System for the creation of OO Enterprise System Architectures in BPR environments”, OOPSLA-97 Conference, 1997. (ftp://www.fti-consulting.com/pub/bpr-papers/oopsla.pdf on 10/4/97).

[Beed97b] Beedle, M., “Pattern Based Reengineering”, Object Magazine, 6(11), pp. 56-70, Jan 1997. (ftp://www.fti-consulting.com/pub/bpr-papers/pbr.pdf on 10-3-97)

[Beri97] Beringer, D., Modelling Global Behaviour with Scenarios in Object-Oriented Analysis, Ph.D. Thesis, Département d'Informatique, Ecole Polytechnique Federale De Lausanne, 1997. (ftp://lglftp.epfl.ch/pub/Papers/beringer-thesis.ps.Z on 1/4/98)

[Berr92] Berre, A., "COOP-an object oriented framework for systems integration", ICSI '92. Proceedings of the Second International, Conference on Systems Integration, (Cat. No.92TH0444-0), IEEE Computer Soc. Press, Los Alamitos, CA, pp. 104-13, 1992.

[BJ94] Beck, K., Johnson, R., “Patterns Generate Architectures”, Proceedings of ECOOP'94, Bologna, Italy, 1994. (ftp://st.cs.uiuc.edu/pub/patterns/papers/patterns-generate-archs.ps on 5/3/96)

[BK86] Bobrow, D., and Katz, R., "Context Structures/Versioning: A Survey". On Knowledge Base Management Systems, Springer-Verlag, New York, pp. 453-460, 1986.

[BM95] Buschman, F., and Meunier, R., “A System of Patterns”, in Pattern Language of Program Design, J. Coplien and D. Schmidt, eds., Addison-Wesley, ISBN 0-201-60734-4, 1995.

[BMB97] Batory, D., Miranker, D., and Brant, D., “Jakarta: A Tool Suite for Constructing Software Generators”, n.pub., n.pag., 1997.
(ftp://ftp.cs.utexas.edu/pub/predator/Joverview.ps on 8/24/97)

[BMRS96] Buschman F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M., Pattern-Oriented Software Architecture: A System of Patterns, John Wiley & SonsChichester, England, ISBN 0-471-95869-7, 1996.

[BN92] Bhansali, S., and Nii, H, “Software design by reusing architectures”, Proceedings of the Seventh IEEE Knowledge-Based Software Engineering Conference, 9/20/92, Mclean, VA, pp. 100-9, 1992.

[BuH89] Burns, H., and Halliburton, R., “Tackling productivity and quality through customer involvement and software technology”, IEEE Global Telecommunications Conference and Exhibition: Communications Technology for the 1990s and Beyond, (Cat. No.89CH2682-3), 11/27/1989, Dallas, TX, Vol.1, pp. 631-5, 1989.

[Buhr97] Buhr, R., “Use case maps (UCMs) Updated: A Simple Visual Notation for Understanding and Architecting the Emergent Behaviour of Large, Complex, Self Modifying Systems”, n.pub., n.pag., 1997.
(http://www.sce.carleton.ca/ftp/pub/UseCaseMaps/ucmUpdate.ps on 4/7/97)

[BvA94] Betlem, B., and van Aggele, R., “An object-oriented framework for production control”, IEE Control ‘94, 3/21/94, pp. 1411-1416, 1994.

[BW88] Bott, M., and Wallis, P., “Ada and software re-use”, Software Engineering Journal, 3(5), pp. 177-83, Sept. 1988.

[CABD94] Coleman, D., Arnold, P., Bodoff, S., Dollin, C., Gilchrist, H., Hayes, F., and Jeremes, P., Object-Oriented Development: The Fusion Method, Prentice Hall, ISBN 0-13-338823-9, 1994.

[Cant92] Cantone, G., “Software factory: modeling the improvement”, Third International IEE Conference on Factory 2000: Competitive Performance Through Advanced Technology, 7/27/92, York, UK, pp. 124-9, 1992.

[Casa90] Casais, E., “Managing Class Evolution in Object-Oriented Systems”, I Centre Universitaire d'Informatique, University of Geneva, pp. 133-195, Jul. 1990.
(ftp://cui.unige.ch/OO-articles/classEvolution.ps.Z on 5/3/96)

[Casa95] Casais, E., “An Experiment in Framework Development - Issues and Results”, in Eduardo Casais (Ed.), Architectures and Processes for Systematic Software Construction, FZI-Publication, Forschungszentrum Informatik Karlsruhe, 1995.
(http://www.fzi.de/prostftp/papers/eiffelFW.ps.Z on 5/3/96)

[Casa95a] Casanave, C., "Business-Object Architectures and Standards", Workshop Report: Business Object Design and Implementation – 10th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications. Addendum to the Proceedings, ACM/SIGPLAN OOPS Messenger, 6(4), Oct. 1995. (http://www.tiac.net/users/jsuth/oopsla/casanpub.pdf on 8/11/96)

[CC93] Cain, B., and Coplien, J., "A role-based Empirical Process Modeling Environment", IEEE Computer Society/ACM SIGGRAPH Second International Conference of the Software Process, 2/25/93, Berlin, Germany, 1993.

[CEW92] Clyde, S., Embley, D., Woodfield, S., "Tunable Formalism in Object-Oriented Systems Analysis: Meeting the Needs of Both Theoreticians and Practitioners", OOPSLA '92, pp. 452-465, 1992.

[Chaf96] Chafi, R., Generic Object Oriented Implementation Design, Ph.D. Dissertation, Illinois Institute of Technology, 1996.

[Chen95] Chen, T., Towards a Construction Theory for Information System Architecure, Ph.D. Thesis, Illinois Institute of Technology, January, 1995.

[CI92] Campbell, R., and Islam, N., "A technique for document the framework of an object-oriented system", Proceedings of the Second International IEEE Workshop on Object Orientation in Operating Systems, 9/24/92, Dourdan, France, pp. 288-300, 1992. (ftp://choices.cs.uiuc.edu/Papers/Conferences/Woos91.frameworks.ps on 5/4/96)

[Cimi92] Cimitile, A., "Towards reuse reengineering of old software", Proceedings of Fourth International IEEE Conference on Software Engineering and Knowledge Engineering, 6/15/92, Capri, Italy, pp. 140-9, 1992.

[CL93] Capretz, L., Lee, P., "Object-oriented design: guidelines and techniques", Information and Software Technology, Vol.35, p. 195-206, Apr. 1993.

[CNM95] Coad, P., North, D., and Mayfield, M., Object Models – Strategies, Patterns, & Applications, Yourdan Press, Englewood Cliffs, NJ, ISBN 0-13-108614-6, 1995.

[Coad92] Coad, P., "Object-Oriented Patterns", Communications of the ACM, Vol.35, Sep. 1992.

[Cock97] Cockburn, A., "Structuring Use cases with goals" JOOP/ROAD, 10(5,7) Sep. 1997 and Nov. 1997. (http://members.aol.com/acockburn/papers/usecases.htm on 11/10/97)

[Coll95] Collins, M., "Iterative Use Case Prototyping", OOPSLA workshop – Requirements Engineering: Use Cases and More, 10/15/95, Austin, Texas, 1995. (http://www.unantes.univ-nantes.fr/usecase/Contributions/mcollins.ps on 2/5/97)

[Come90] Comer, E., "Domain analysis: a systems approach to software reuse", Proceedings. IEEE/AIAA/NASA 9th Digital Avionics Systems Conference, 10/15/90, Virginia Beach, VA, pp. 224-9, 1990.

[Cope94] Coplien, J., "A Development Process Generative Pattern Language", First Annual Conference on the Pattern Languages of Programming, Monticello, Illinois, August 1994.

[Cope95] Coplien, J., Pattern Language of Program Design, J. Coplien and D. Schmidt, eds., Addison-Wesley, ISBN 0-201-60734-4, 1995.

[Cox87] Cox, B., "Building malleable system from software 'chips'", Computerworld. pp. 59-68, Mar. 1987.

[CR92] Cybulski, J., and Reed, K., "A hypertext based software-engineering environment", IEEE Software, 9(2) pp. 62-8, Mar. 1992.

[CWC94] Cima, A., Werner, C., and Cerqueira, A., "The design of object-oriented software with domain architecture reuse", Proceedings. Third International Conference on Software Reuse: Advances in Software Reusability, 11/1/94, Rio de Janeiro, Brazil, p.178-87, IEEE Computer Soc. Press, ISBN: 0 8186 6632 3, 1994.

[Davi92] Davis, M., "STARS Reuse Maturity Model: Guidelines for Reuse Strategy Formulation", WISR 5 - Fifth Annual Workshop on Software Reuse, 1992.

[dCLF93] de Champeaux, D., Lea, D., and Faure, P., Object Oriented System Development, Addison-Wesley, Reading Mass, 1993.

[DDB93] Dubois, E., Du Bois P., and Petit, M., "O-O Requirements Analysis: an Agent Perspective", Proceedings ECOOP '93 – Object-Oriented Programming 7th European Conference, Kaiserland, Germany, 7/26/1993, pp. 458-81, 1993.

[Digr95] Digre, T., "Business Application Components", Workshop Report: Business Object Design and Implementation. 10th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications. Addendum to the Proceedings, ACM/SIGPLAN OOPS Messenger 6(4), pp. 170-175, Oct. 1995.

[DMR94] DeBaud, J., Moopen, B., and Rugaber, S., "Domain Analysis and Reverse Engineering", Proceedings of the 1994 International Conference on Software Maintenance, pp. 326-335, 1994. (ftp://ftp.cc.gatech.edu/pub/groups/reverse/repository/domain.ps on 5/3/96)

[DR95] DeBaud J., Rugaber, S., "A Software Re-Engineering Method using Domain Models", IEEE Proceedings. International Conference on Software Maintenance, 10/17/95, IEEE Computer Soc. Press, Los Alamitos, CA, pp. 204-13 ISBN: 0 8186 7141 6, 1995.

[DW97] D'Souza, D., and Wills, A., Component-Based Development Using Catalysis, Draft v 0.8, ICON Computing, 1997. (http://www.iconcomp.com on 3/3/97)

[DWMW96] Davies, R., May, P., Wardell, D., and Wooding, T., "Techniques for developing reusable business components", JOOP, 9(7), , pp. 40-43, Nov-Dec. 1996.

[EDF96] Ecklund, Jr, E., Delcambre, L., and Freiling, M., "Change Cases: Use Cases that Identify Future Requirements", OOPSLA: The First Eleven Years Conference Proceedings 1986-1996. CD-ROM. ACM Press, New York, 1997.

[EK87] Elam, J., and Konsynski, B., "Using Artificial Intelligence Techniques to Enhance the Capabilities of Model Management Systems", Decision Sciences, Vol.18, pp. 4487-501, Sum. 1987.

[EvBD92] Erradi, M.; von Bochmann, G.; Dssouli, R., "A framework for dynamic evolution of object-oriented specifications", IEEE Conference on Software Maintenance 11/9/92, Orlando, FL, n.pag., 1992.

[Fall93] Falla, M., "Using formal English for object-oriented system description", IEE Colloquium on Object Oriented Development, (Digest No.007) 1/14/93, London, UK, p: 3/1-3, 1993.

[FCS97] Fingar, P., Clarke, J., and Stikeleather, J., “The business of distributed object computing”, Object Magazine, 7(2), pp. 28-33, Apr. 1997.

[Fink88] Finkelstein, A., “Re-use of formatted requirements specifications”, Software Engineering Journal, 3(5), pp. 186-97, Sep. 1988.

[FL94] Fridman N., and Lieberherr K., “Reuse of Adaptive Software through Opportunistic Parameterization” 16 p. n.pub., 1994 (ftp://ftp.ccs.neu.edu/pub/research/demeter/documents/papers/FL94-opportunistic-params.ps on 5/3/96)

[FNP92] Fugini, M., Nierstrasz, O., and Pernici, B., “Application Development through Reuse: the Ithaca Tools Environment”, SIGOIS Bulletin, 13(2), pp. 38-47, Aug. 1992. (ftp://cui.unige.ch/OO-articles/appDevThroughReuse.ps.Z on 5/3/96)

[Fowl96] Fowler, M., Analysis Patterns : Reusable Object Models, Addison-Wesley, ISBN: 0201895420, 1996.

[Fowl97] Fowler, M., “Recurring Events for Calendars”, n.pub., n.pag., 1997. (http://www.awl.com/cseng/titles/0-201-895420/events2-1.htm on 11/7/97)

[Fran90] Franke, D., “Imbedding rule inferencing in applications”, IEEE Expert, 5(6), pp. 8-14, Dec. 1990.

[Fran97] Franz, Inc. “Building Knowledge-Based Applications with Dynamic Object Technology”, White Paper. Franz, Inc., Berkeley, CA, n.pub., n.pag., 1997.

[Gant90] Ganti, M., Goyal, P., Nassif, R., and Podar, S., “An object-oriented application development environment”, COMPCON Spring '90: Thirty-Fifth IEEE Computer Society International Conference. Intellectual Leverage. Digest of Papers, 2/26/90, San Francisco, CA, pp. 348-55, 1990.

[GE97] Gale T., and Eldred, J., “The Abstract Business Process”, Object Magazine, 6(11), pp. 21-37, Jan. 1997.

[GHJV94] Gamma, E., Helm, R., Johnson, R., and Vlissides, J., Design Pattterns: Elements of Reusable Software, Addison-Wesley ISBN 0-201-63361-2, 1994.

[Gilb96] Gilbert, M., “Is Object COBOL a framework for reuse”, Object Magazine, 6(3), pp. 42-47, May 1996.

[GK92] Gall, H., and Klosch, R., “Reuse engineering: software construction from reusable components”, IEEE Proceedings. The Sixteenth Annual International Computer Software and Applications Conference, 9/2192, Chicago, IL, p: 79-86, 1992.

[GK96] Griss M., and Kessler, R., “Building Object-oriented Instrument Kits”, Object Magazine, 6(2), pp. 71-81, Apr. 1996.

[GKS92] Goul, M., Kuo, C., and Sandman, T., “Towards synergizing the active object, software maintenance, and algorithm synthesis metaphors for integrated modeling environments”, ACM/IEEE Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences, 1/7/92, Kauai, HI, Vol.3, pp. 437-48, Jan. 1992.

[Glin95] Glinz, M., “An Integrated Formal Model of Scenarios Based on Statecharts”, Proceedings of the 5th European Software Engineering Conference, Sitges, Spain, n.pag., 1995.

[GM95] Gangopadhyay D., and Mitra, S., "Understanding Frameworks by Exploration of Exemplars", Proceedings of 7th International Workshop on Computer Aided Software Engineering, (CASE-95), IEEE Computer Society Press, ISBN 0-8186-7078-9, pp. 90-99, Jul. 1995. (ftp://www.pt.hk-r.se/~michaelm/Case95.Final.ps on 11/21/97)

[Gold85] Goldstein, R., Database - Technology and Management, John Wiley & Sons, New York, 1985, ISBN 0-471-88737-4, 1985.

[Goma92] Gomaa, H., "An object-oriented domain analysis and modeling method for software reuse", ACM/IEEE Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences, 1/7/97, Kauai, HI, Vol.2, pp. 46-56, 1992.

[Goma93] Gomaa, H., "A reuse-oriented approach for structuring and configuring distributed applications", Software Engineering Journal, 8(2), pp. 61-71, Mar. 1993.

[Gott97] Gottesdiener, E., "Business Rules show power, promise", Application Development Trends, 4(3), pp36-53, Mar. 1997.

[Grah94] Graham, I., Migrating to Object Technology, Addison-Wesley, ISBN 0-201-59389-0, 1994.

[Gris95] Griss, M., "Packaging Software Reuse Technologies as Kits", Object Magazine, 5(6), pp. 80-81,89, Oct. 1995.

[GS88] Garg, P., and Scacchi, W., "A hypertext system to manage software life cycle document", Proceedings of the Twenty-First Annual Hawaii International Conference on System Sciences, (Cat. No.88TH0212-1), Vol.II. p: 337-46, 1988.

[GTC+90] Gibbs, S., Tsichritzis, D., Casais, E., Nierstrasz O., and Pintado, X., "Class Management for Software Communities", Communications of the ACM, pp. 90-103, Sep. 1990.

[Hans95] Hansen, T., "Requirements definition and verification through use case analysis and early prototyping", OOPSLA workshop – Requirements Engineering: Use Cases and More, 10/15/95, Austin, Texas, 1995. (http://www.unantes.univ-nantes.fr/usecase/Contributions/toddHansen.ps on 2/5/97)

[Harw97] Harwood, R., "Use case formats: Requirements, analysis and design", JOOP, 9(8), pp. 54-57, Jan. 1997.

[HBPP95] Hertha, W., Bennett, J., Post, F., Page, I., "An Architecture Framework: From Business Strategies to Implementation", Workshop Report: Business Object Design and Implementation. 10th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications. Addendum to the Proceedings. ACM/SIGPLAN OOPS Messenger, 6(4), Oct. 1995. (http://www.tiac.net/users/jsuth/oopsla/herthpub.pdf on 8/7/96)

[HC91] Hayes F., and Coleman, D., "Coherent Models for Object-Oriented Analysis", OOPSLA '91, pp.171-183, 1991.

[Hend97] Hendeerson-Sellers, B., "Choosing between UML and OPEN", n.pub., n.pag., 1997. http://www.csse.swin.edu.au/cotar/OPEN/CHOOSING/choosing.html on 11/14/97)

[Herb95] Herbst, H., "A Meta-Model for Business Rules in Systems Analysis", Proceedings of the Seventh Conference on Advanced Information Systems Engineering, Berlin. Springer, pp. 186 – 199, 1995. (ftp://ftp.ie.iwi.unibe.ch/public/documents/papers/caise95.eps on 1/12/98)

[HH95] Hay, D., Healy, K., GUIDE Business Rules Project - Final Report, November 7, 1995 (http://www.guide.org/ap/apbrules.pdf on 3/29/97)

[HHMV93] Helm, R., Huynh, T., Kim Marriott, and John Vlissides, “An Object-Oriented Architecture for Constraint-Based Graphical Editing”, Advances in Object-Oriented Graphics II, Springer Verlag, n.pag., 1993.

[Hink94] Hinkle, B., “Reflective Programming in Smalltalk-80”, Tutorial, OOPSLA '94, n.pag., 1994.

[HJS92] Hartmann, T., Junglclaus, R., and Saale, G., “Aggregation in a Behavior Oriented Object Model”, ECOOP 92, 6/29/97, Utrecht, The Netherlands, pp. 57-77, 1992.

[HM95] Herbst, H., and Myrach, T., “A Repository System for Business Rules”, Proceedings of the Sixth IFIP TC-2 Working Conference on Data Semantics, London. Chapman & Hall 1996. (ftp://ftp.ie.iwi.unibe.ch/public/documents/papers/ds-6-95.eps on 1/12/98)

[Holl92] Holland, I., “Specifying reusable components using Contracts”, ECOOP '92, 6/27/92, Utrecht, The Netherlands, pp. 288-308, 1992.

[Holl94] Hollingsworth, D., Workflow Management Coalition – The Workflow Reference Model, Document Number TC00-1003, Issue 1.1, Workflow Management Coalition, Brussels, Belgium, 1994, revised 1997. (ftp://ftp.aiai.ed.ac.uk/pub/projects/WfMC/refmodel/rmv1-16.pdf on 10/17/97)

[Horn89] Horn, R., Mapping Hypertext: Analysis, Linkage, and Display of Knowledge for the Next Generation of On-Line Text and Graphics, Lexington Institute, Lexington, MA, 1989.

[HSFG97] Henderson-Sellers, B., Firesmith, D., and Graham, I., “OML Metamodel: Relationships and state modeling”, JOOP, 10(1), Mar-Apr. 1997.

[HSGK94] Hsia, P., Samuel, J., Gao, J., and Kung, D., “Formal Approach to Scenario Analysis”, IEEE Software, 11(2), Mar. 1994.

[HTP89] Heisler, K., Tsai, W., and Powell, P., “An object-oriented maintenance-oriented model for software”, Thirty-Fourth IEEE COMPCON Spring '89. Digest of Papers – Computer Society International Conference: Intellectual Leverage, (IEEE Cat. No.89CH2686-4) 2/27/89, San Francisco, CA, pp. 248-53, 1989.

[Hurs95] Hursch, W., Maintaining Behavior and Consistency of Object-Oriented Systems during Evolution, Ph.D. Thesis, College of Computer Science, Northeastern University, Boston, MA, 331 pages, August 1995.

[ILOG97] ILOG, ILOG Rules White Paper, ILOG, Inc., Cambridge, MA, 1997. (http://www.ilog.com/papers/control/ILOG_Rules_WP.pdf on 7/10/97)

[Jack94] Jackson, M., “Problems, Methods and Specialisation”, Special Issue of SE Journal on Software Engineering in the Year 2001, n.pag., 1994.

[Jaco95] Jacobson, I., “Modeling with Use Cases: Formalizing use-case modeling”, JOOP 8(3), Jun. 1995.

[Jans95] Jansson, P., “Use Case Analysis with Rational Rose”, Chapter 4, Rational Rose Application Notes, Software Release 3.0, Rational Software Corp, Santa Clara CA, 1995.

[JB93] Johnson, W., Benner, K., and Harris, D., “Developing formal specifications from informal requirements”, IEEE Expert, 8(4), pp. 82-90, Aug. 1993.

[JBJE95] Jacobson, I., Bylund, S., Jonsson, P., and Ehneboom, S., "Modeling With Use Cases: Using contracts and use cases to build pluggable architectures", JOOP 8(2), pp. 1-24,76, May 1995. (http://www.unantes.univ-nantes.fr/usecase/JOOP.ps on 2/5/97)

[JCJO92] Jacobson, I., Christenson, M., Jonsson, P., Overgaard, G., Object-oriented software engineering : a use case driven approach, ACM Press, Reading, Mass. Addison-Wesley, 1992.

[JEJ94] Jacobson, I., Ericsson, M., and Jacobson, A., The Object Advantage – Business Process Reengineering with Object Technology, ACM Press, ISBN 0-201-42289-1, 1994.

[JF88] Johnson R., and Foote. B., “Designing reusable classes”, JOOP, 1(2), pp. 22-35, 1988.

[JGJ97] Jacobson, I., Griss, M., Jonsson, P., Software Reuse - Architecture, Process and Organization for Business Success, ACM Press, New York, ISBN 0-201-92476-5, 1997.

[JO93] Johnson R., and Opdyke. W., “Refactoring and Aggregation”, S. Nishio and A. Yonezawa, editors, International Symposium on Object Technologies and Advanced Software (ISOTAS), 11/93, Kanazawa, Japan, JSSST, Lecture Notes in Computer Science, Springer Verlag, Vol. 742, pages 264-278, 1993.

[John92] Johnson, R., “Document Frameworks using Patterns”, OOPSLA '92, pp. 63-76, 1992.

[John93] Johnson, R., “How to Design Frameworks”, Tutorial Notes, OOPSLA '93, Washington, 1993, (ftp://st.cs.uiuc.edu/pub/papers/frameworks/OOPSLA93-frmwk-tut.ps on 4/20/96)

[Jone91] Jones, S., Text and Context: Document Storage and Processing, Springer-Verlag, London, 1991.

[JP93] Johnson R., and Palaniappan, M., “MetaFlex: A Flexible Metaclass Generator”, ECOOP '93 - Object-Oriented Programming 7th European Conference, Kaiserland, Germany, 7/26/93, pp. 502-527, 1993.

[JR91] Johnson R., and Russo, V., “Reusing Object-Oriented Design”, University of Illinois, Technical Report UIUCDCS 91-1696, 1991.

[KAC86] Katz, R., Anwarrudin, M., and Chang, E., “Organizing A Design Database Across Time”. On Knowledge Base Management Systems, Springer-Verlag, New York, pp. 287-296, 1986.

[Kain96] Kain, J., “Components: The Basics”, Object Magazine, 6(2), pp. 65-59, Apr. 1996.

[KCM91] Kaplan, S., Carroll A., and MacGregor, K., “Supporting Collaborative Processes with ConversationBuilder”, ACM SIGOIS Conference on Organizational Computing Systems, SIGOIS Bulletin, 12(2,3) pp. 69-79. 1991.

[KHT95] Kilov, H., Harvey, B., and Tyson, K., “Semantic integration in complex systems: collective behavior in business rules and software transactions”, OOPSLA '95 - Addendum to the Proceedings, OOPSLA: The First Eleven Years Conference Proceedings 1986-1996. CD-ROM. ACM Press, New York, 1997.

[KL95] Koch G., and Loney, K., Oracle - The Complete Reference, Third Edition, Osborne McGraw-Hill, Berkeley, California, ISBN 0-07-882097-9, 1995.

[KM95] Koskimies K., and Mossenbock, H., “Designing a Framework by Stepwise Generalization”, 5th European Software Engineering Conference, Barcelona. Lecture Notes in Computer Science, Springer-Verlag, Vol. 989, pp. 479-497, 1995. (http://oberon.ssw.uni-linz.ac.at/Frameworks.ps.Z on 5/3/96)

[KMJ97] Kendall, E., Malkoun, M., and Jiang. C., “The application of object oriented analysis to agent-based system”, JOOP, 9(9), pp. 56-63, Feb. 1997.

[Know95] Knowles, N., “Reuse Cases?”, OOPSLA workshop - Requirements Engineering: Use Cases and More, 10/15/95, Austin, Texas, 1995. (http://www.unantes.univ-nantes.fr/usecase/Contributions/nickKnowles.html on 2/5/97)

[Koen95] Koenig, A., “Patterns and anti-patterns”, JOOP, pp. 46-48, Mar-Apr. 1995.

[Kors96] Korson, T., “Managing Reuse: Applying the Law of Gravity”, Object Magazine, 6(2), pp. 34-36, Apr. 1996.

[KPW97] Kosters, G., Pagel, B., and Winter, M., “Coupling Use Cases and Class Models”, Proc. of the BCS-FACS/EROS workshop on “Making Object Oriented Methods More Rigorous, 6/24/97, Imperial College, London, pp. 27-30, 1997. (ftp://ftp.fernuni-hagen.de/pub/fachb/inf/pri3/papers/winter/RoomAbstract.ps.gz on 8/23/97)

[Krue92] Krueger, C., “Software Reuse”, ACM Computing Surveys, 24(2), Jun. 1992.

[KS94] Kimbler, K., and Sobirk, D., “Use Case Driven Analysis of Feature Interactions”, Feature Interactions in Telecommunications Systems, IOS Press, 1994. (http://www.tts.lth.se/Personal/daniels/papers/fiw94.ps 1/4/98)

[LBS90] Lieberherr, K., Bergstein, R., and Silva-Lepe, I., “Abstraction of object-oriented data models” Proc. Int. Conf. on Entity-Relationship Approach, Lausanne, Switzerland, 1990, H. Kangass (ed.) (Elsevier), pp.88-102, 1990.

[LBS91] Lieberherr, K., Bergstein, P., Silva-Lepe, I., “From objects to classes: algorithms for optimal object-oriented design”, IEE Colloquium on ‘Applications and Experience of Object-Oriented Design’ (Digest No.018), Software Engineering Journal, 6(4), pp. 205-28, Jul. 1991.

[LH89] Lieberherr, K.J., and Holland, I., “Tools for preventive software maintenance”, IEEE Conference on Software Maintenance, (Cat. No.89CH2744-1) 10/16/89, Miami, FL, pp. 2-13, Oct. 1989.

[LH92] Lano, K., and Haughton, H., “Extracting design and functionality from code”, Proceedings. Fifth International Workshop on Computer-Aided Software Engineering, (Cat. No.92CH3166-6), IEEE Computer Soc. Press, Los Alamitos, CA, pp. 74-82, 1992.

[LHR88] Lieberherr, K., Holland, I., and Riel, A. “Object-oriented programming: An objective sense of style”, OOPSLA '88, pp. 323-334, 1988.

[Lieb95] Lieberherr, K., “Report: OOPSLA'95 Workshop on Adaptable and Adaptive Software”, OOPSLA: The First Eleven Years Conference Proceedings 1986-1996. CD-ROM. ACM Press, New York, 1995.

[Lieb96] Lieberherr, K., Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns, PWS Publishing Company, ISBN 0-534-94602-X, 1996.

[Lips96] Lipshutz, M., “Training Materials For The Software Technology For Adaptable, Reliable Systems (STARS)”, STARS-PA19-S009/002/00, Electronic Systems Center, Air Force Material Command, USAF Hanscom AFB, 1996. (http://www.asset.com/WSRD/ASSET/A/1181/ASSET_A_1181.tar.gz on 8/24/97)

[LK92] Li, H., and van Katwijk, J., “Issues concerning software reuse-in-the-large”, ICSI '92. IEEE Proceedings of the Second International Conference on Systems Integration, (Cat. No.92TH0444-0) 6/15/92, Morristown, NJ, pp. 66-75, 15-18, 1992.

[LK94] Lajoie, R., and Keller, R., “Design and Reuse in Object-Oriented Frameworks: Patterns, Contracts, and Motifs in Concert”, Proceedings of the 62nd Congress of the Association Canadienne Francaise pour l'Avancement des Sciences, 5/94, Montreal, Canada, 1994. (ftp://st.cs.uiuc.edu/pub/patterns/papers/acfas.ps on 5/3/96)

[LL92] Lin F., and Langsner, R., “Integrating CASE tools with knowledge-base by object orientation”, IEEE Proceedings. ICCI '92. Fourth International Conference on Computing and Information, 5/28/92, Toronto, Canada, pp. 325-8, 1992.

[LS94] Lortz, V., Shin, K., “Combining Contracts and Exemplar-Based Programming for Class Hiding and Customization”, OOPSLA '94, pp. 452-467, 1994.

[Luba91] Lubars, M., “Reusing designs for rapid application development”, ICC 91. IEEE International Conference on Communications Conference Record, 6/23/91, Denver, CO, Vol.3, pp. 1515-19, 1991.

[Mart95] Martin, R., “Discovering Patterns in Existing Code”, Pattern Language of Program Design, J. Coplien and D. Schmidt, eds., Addison-Wesley, ISBN 0-201-60734-4, 1995.

[McIl69] McIlroy, M., “Mass-Produced Software Components” Software Engineering Concepts and Techniques: 1968 NATO Conference on Software Engineering. Burton, Naur, Randell, eds. Petrocelli/Charter, pp. 88-98, New York, 1969.

[MDK96] McGregor, J., Doble, J., and Keddy, A., “A Pattern For Reuse”, Object Magazine, 6(2) pp. 38-47, Apr. 1996.

[Mend95] Mendoza-Grado, V., “Formal Verification of Use Cases”, OOPSLA workshop – Requirements Engineering: Use Cases and More, 10/15/95, Austin, Texas, 1995. http://www.unantes.univ-nantes.fr/usecase/Contributions/mendoza.ps on 2/5/97)

[MK92] Mittermeir, R., and Kofler, E., “Layered specifications to support reusability and integratibility”, ICSI '92. IEEE Proceedings of the Second International Conference on Systems Integration, 6/15/92, Morristown, NJ, p: 699-708, 1992.

[ML97] Mitchell I., and Lecoeuche, H., “On an improved approach to the elicitation of O-O state machines by use-case”, JOOP, 9(9), pp. 52-55, Feb. 1997.

[MMKP91] Moser, K., Mazzola, D., Keim, R., and Philippakis, A., “Modeling the information systems architecture: an object-oriented approach”, Proceedings of the Twenty-Fourth Annual Hawaii, International Conference on System Sciences (Cat. No.91TH0350-9) Vol. 4, IEEE Comput. Soc. Press, Los Alamitos, CA, pp. 83-92, 1991.

[MMT94] Menga, G., Messina, G., and Tricomi, G., "An object-oriented framework for enterprise modeling", IAS '94. IEEE Conference Record of the 1994 Industry Applications Conference Twenty-Ninth IAS Annual Meeting, 10/2/94, Denver, CO, IEEE, New York, NY, Vol.3, ISBN: 0 7803 1993 1, pp. 1879-86, 1994.

[MN93] Murphy, G., and Notkin, D., "The Interaction Between Static Typing and Frameworks", University of Washington, Department of Computer Science and Engineering, Technical Report 93-09-02, 18 p, 1993. (ftp://cs.washington.edu/tr/1993/09/UW-CSE-93-09-02.PS.Z on 5/3/96)

[Moor96] Moore, I., "Automatic inheritance hierarchy restructuring and method refactoring", OOPSLA: The First Eleven Years Conference Proceedings 1986-1996. CD-ROM. ACM Press, New York, 1996.

[MS91] Moffett, J., Morris and Sloman, S., "The Representation of Policies as System Objects", 1991 ACM SIGOIS Conference on Organizational Computing Systems, SIGOIS Bulletin, 12(2,3), pp. 171-184, 1991.

[MTMS92] Mark, W., Tyler, S., McGuire, J., and Schlossberg, J., "Commitment-based software development", IEEE Transactions on Software Engineering, 18(10), pp. 870-85, Oct. 1992.

[MWFF92] Medina-Mora, R., Winograd, T., Flores, R., and Flores, F., "The Action Workflow Approach to Workflow Management Technology", CSCW 92 Proceedings, November 1992.

[Nass91] Nassiff, E., The Hierarchical Policy Model, MS Thesis, Illinois Institute of Technology, 1991.

[NC89] Nunamaker, Jr., J., and Chen, M., "Software productivity: a framework of study and an approach to reusable components", IEEE Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences, 1/3/89, Kailua-Kona, HI, Vol.II: Software Track p: 959-68, 1989.

[Neig84] Neighbors, J., "The DRACO approach to constructing software from reusable components", IEEE Transactions Software Engineering, 10(5), pp. 564-574, Sep. 1984.

[NGT92] Nierstrasz, O., Gibbs S., and Tsichritzis, D., "Component-Oriented Software Development", Communications of the ACM, Vol. 35, p. 63-74, Sep. 1992.

[OH95] Ossher, H. and Harrison, W., "Developing Adapatable Software with Subject-Oriented Programming", OOPSLA'95 Adaptable and Adaptive Software Workshop, 1995. (http://www.ccs.neu.edu/research/demeter/adaptable-systems/Ossher.ps on 3/29/97)

[OJ93] Opdyke W., and Johnson. R., "Creating abstract superclasses by refactoring", Proceedings of CSC '93: The ACM 1993 Computer Science Conference, February 1993.

[Oops95] OOPSLA "Systematic Software Reuse: Objects and frameworks are not ecough (Panel)", OOPSLA '95, Austin, Texas, 1995, p. 281, 1995.

[Oops95b] OOPSLA, "PANEL: Objects and Domain Engineering", OOPSLA '95, Austin, Texas, pp. 333-336, 1995.

[Opdy92] Opdyke, W., Refactoring Object-Oriented Frameworks, Ph.D. Thesis, University of Illinois at Urbana-Champaign 1992.

[PA95] Poirier S., and Ashford, C., "Semantics: the key to interoperability", in Workshop Report: Business Object Design and Implementation. 10th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications. Addendum to the Proceedings. ACM/SIGPLAN OOPS Messenger, 6(4), Oct. 1995. (http://www.tiac.net/users/jsuth/oopsla/poirier.html on 8/11/96)

[PBC92] Patel, S., Baxter, R., Chu, W., Sayrs, B., and Sherman, S., "Automated assessment of program and system quality", IEEE Proceedings of the Second Symposium on Assessment of Quality Software Development Tools, 5/27/92, New Orleans, LA, pp. 112-19, 1992.

[PC93] Poulin, J., and Caruso, J., "Determining the value of a corporate reuse program", IEEE Proceedings First International Software Metrics Symposium, 5/21/93, Baltimore, MD, p: 16-27, 1993.

[PCCW93] Paulk, M., Curtis, B., Chrissis, M., Weber, C., Capability Maturity Model for Software, Version 1.1, Technical Report, CMU/SEI-93-TR-024, Software Engineering Institute, Carnegie Mellon University, Feb. 1993. (http://www.asset.com/WSRD/ASSET/A/1610/elements/tr24.93.pdf on 3/6/98)

[Pern90] Pernici, B., "Objects with Roles", 1990 ACM SIGOIS Conference on Office Automation Systems, SIGOIS Bulletin 11(2,3) 1990, pp.205-215, 1990.

[Pree94] Pree, W., Design Patterns for Object-Oriented Software Development, ACM Press, New York, 1994, ISBN 1-201-42294, 1994.

[Pree96] Pree, W., "Frameworks - Past, Present, Future", Object Magazine, 6(3), pp. 24-36, May 1996.

[Prie93] Prieto-Diaz, R., "Status report: software reusability", IEEE Software, 10(3), pp. 61-6, May 1993.

[PS94] Peterson, S., Stanley, Jr., J., "Mapping a Domain Model and Architecture to a Generic Design", Technical Report CMU/SEI-94-TR-8 ESC-TR-94-008, The Software Engineering Institute, Carnegie Mellon University, 1994. (http://www.sei.cmu.edu/pub/documents/94.reports/pdf/tr08.94.pdf on 5/1/97)

[PT90] Pintado, X., and Tsichritzis, D., "SaTellite: A Visualization and Navigation Tool for Hypermedia", 1990 ACM SIGOIS Conference on Office Automation Systems, SIGOIS Bulletin 11(2,3), pp.271-280, 1990.

[RAB96] Regnell, B., Andersson, M., and Bergstrand, J., "A Hierarchical Use Case Model with Graphical Representation", Proceedings of Second International Symposium on Engineering Computer-Based Systems, Friedrichshafen, Germany, IEEE Computer Society Press, ISBN 0-8186-7355-9, March 1996, pp. 270-277. (http://www.tts.lth.se/Personal/bjornr/Papers/ECBS96.ps 1/4/98)

[Rama95] Ramackers, G., "Object Business Modelling, requirements and approach", in Workshop Report: Business Object Design and Implementation. 10th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications. Addendum to the Proceedings. ACM/SIGPLAN OOPS Messenger, 6(4), Oct. 1995. (http://www.tiac.net/users/jsuth/oopsla/ramack.html on 5/2/96)

[Raws95] Rawsthorne, D., “Transaction Based Analysis Capturing Functional Requirements Through Object Interactions”, OOPSLA workshop – Requirements Engineering: Use Cases and More, 10/15/95, Austin, Texas, 1995. (http://www.unantes.univ-nantes.fr/usecase/Contributions/drawstho.ps on 2/5/97)

[RB95] Roberts, D., and Brandt, J., “Supporting Framework Evolution with Refactorings”, Seventh Annual Workshop on Institutionalizing Software Reuse (WISR), 8/28/95, St. Charles, Illinois, 1995. (ftp://gandalf.umcs.maine.edu/pub/WISR/wisr7/proceedings/txt/roberts.txt on 1/4/98)

[RD97] Regnell, B., and Davidson, A., “From Requirements to Design with Use Cases - Experiences from Industrial”, REFSQ'97: 3rd Intl Workshop on Requirements Engineering - Foundation for Software Quality, 6/20/97, Barcelona, Presses universitaires de Namur, ISBN 2-87037-239-6, pp. 205-222, 1997. (http://www.tts.lth.se/Personal/bjornr/Papers/REFSQ97.ps on 1/4/98)

[Ried90] Riedesel, J., “An object oriented model for expert system shell design”, IEEE Ninth Annual International Phoenix Conference on Computers and Communications, (Cat. No.90CH2799-5) 3/21/90, Scottsdale, AZ, pp. 699-705, 1990.

[Rieh95] Riehle, D., “A Pattern Language for Tool Construction and Integration Based on the Tools and Materials Metaphor”, Pattern Language of Program Design, J. Coplien and D. Schmidt, eds., Addison-Wesley, ISBN 0-201-60734-4, 1995.

[RKW95] Regnell, B., Kimbler, K., and Wesslen, A., “Improving the Use Case Driven Approach to Requirements Engineering”, Proceedings of Second International Symposium on Requirements Engineering, York, UK, IEEE Computer Society Press, ISBN 0-8186-7017-7, pp. 40-47, 1995. (http://www.tts.lth.se/Personal/bjornr/Papers/tts-94-24.ps on 1/4/98)

[Ross97] Ross, R., The Business Rule Book - Classifying, Defining and Modeling Rules, Second Edition” Database Research Group, Boston, MA, 1997.

[RWM+92] Rada, R., Weigang Wang, Mili, H., Heger, J., Scherr, W., “Software reuse: from text to hypertext”, Software Engineering Journal, 7(5), pp. 311-21, Sep. 1992.

[SC95] Silva, M., and Carlson, C., “MOODD, a method for object-oriented database design”, Data and Knowledge Engineering, Vol. 17, pp. 159-181, 1995.

[SCCB95] Shindyalow, I., Chang, W., Cooper, J., and Bourne, P., “Design and use of a software framework to obtain information derived from macromolecular structure data”, IEEE Proceedings of the 28th Annual Hawaii International Conference of System Sciences, 1995.

[Schm95] Schmid, H., “Creating the Architecture of a Manufacturing Framework by Design Patterns”, OOPSLA '95, Austin, Texas, pp.370-384, 1995.

[Schw95] Schwaber, K., “SCRUM Development Process”, Workshop Report: Business Object Design and Implementation. 10th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications. Addendum to the Proceedings. ACM/SIGPLAN OOPS Messenger 6(4), Oct. 1995. (http://www.tiac.net/users/jsuth/oopsla/schwapub.pdf on 8/7/96)

[Seng92] Senge, P., “Building Learning Organization”, Journal for Quality and Participation, Mar. 1992.

[Shaw94] Shaw, M., “Patterns for Software Architecutres, Pattern Language of Program Design, J. Coplien and D. Schmidt, eds., Addison-Wesley, ISBN 0-201-60734-4, 1994.

[Silv94] Silva-Lepe, I., Techniques for reverse-engineering and Re-engineering into the object-oriented Paradigm, Ph.D. Thesis, June 1994.

[Silv95] Silva, M., A/OODBMT - An Active Object-Oriented Database Modeling Technique, Ph.D. Thesis, Illinios Institute of Technology, Computer Science Department of Illinois Institute of Technology, 1995.

[Sing96] Singhal. S., A Programming Language for Writing Domain-Specific Software System Generators, Ph.D. Dissertation. Department of Computer Sciences, University of Texas at Austin, September 1996.
(ftp://ftp.cs.utexas.edu/pub/predator/vivek-thesis.ps.Z on 8/24/97)

[SMLD96] Steyaert, P., Lucas, C., Mens, K., and D'Hondt, T., “Reuse contracts: managing the evolution of reusable assets”, OOPSLA: The First Eleven Years Conference Proceedings 1986-1996. CD-ROM. ACM Press, New York, 1996.

[SS97] Swaminathan V., and Storey, J., “Domain specific frameworks”, Object Magazine, 7(2), pp. 53-57, Apr. 1997.

[SSP95] Stewarti, S., and St. Pierre, J., “Experiences with a Manufacturing Framework”, Workshop Report: Business Object Design and Implementation. 10th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications. Addendum to the Proceedings. ACM/SIGPLAN OOPS Messenger 6(4), October, 1995.
(http://www.tiac.net/users/jsuth/oopsla/stewapub.pdf on 5/2/96)

[Stik96] Stikeleather, J., “The importance of Architecture”, Object Magazine, 6(2), pp. 20-24, Apr. 1996.

[Su91] Su, L., Conceptual Modeling Framework and Semantic Query Interface for Multi-Model Information Systems, Ph.D. Thesis, Illinois Institute of Technology, September 1991.

[Suth95] Sutherland, J., “Business Object Design and Implementation”, 10th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications Addendum to the Proceedings. ACM/SIGPLAN OOPS Messenger, 6(4), pp. 170-175, Oct. 1995.

[Swan96] Swanstrom. E., “Object Oriented Change and Learning Methodology” Arthur D. Little, White Paper, 1996.

[Tali93] Taligent Inc., “Leveraging Object-Oriented Frameworks”, A Taligent White Paper, 1993.
(http://www.taligent.com/leveraging-oofw.html on 5/10/96)

[Tali94] Taligent Inc., “Building Object-Oriented Frameworks”, A Taligent White Paper, 1994.
(http://www.taligent.com/building-oofw.html on 5/10/96)

[Thom95] Thomas, L., “Animating Use Cases”, OOPSLA workshop - Requirements Engineering: Use Cases and More, 10/15/95, Austin, Texas, 1995.
(http://www.unantes.univ-nantes.fr/usecase/Contributions/lThomas.ps on 2/5/97)

[TT92] Tamai, T., and Torimitsu, Y., “Software lifetime and its evolution process over generations”, IEEE Conference on Software Maintenance, , 11/9/92, Orlando, FL, pp. 63-9, 1992.

[UM91] Umphress D., and March, S., “Object-Oriented Requirements Determination”, JOOP, 1991 Focus on Analysis and Design, pp. 35-40, 1991.

[UML97] Unified Modeling Language, Version 1.1, Rational Software, Santa Clara, CA, 1997. (http://www.rational.com/uml on 9/28/97)

[Venk96] Venkatesam, S., “Software Cost Accounting - A New Dimension”, National Institute of Public Enterprise Software Engineering Symposium, Hyderabad, India, 8/23/96, 1996.

[VJK96] Vaishnavi V., and Joosten, S., Bill Kuechlerin, “Modeling Workflow Management Systems Using Smart Objects”, Proceedings of the NSF Workshop on Workflow and Process Automation in Information Systems, 5/96, Athens, GA, 1996. (http://www.cis.gsu.edu/~vvaishna/object_research/projects/smart_object/allsec3.htm on 11/22/97)

[Vlis95] Vlissides, J., “Reverse Architecture”, Position Paper, Dagstuhl Seminar, 9508, IBM T.J. Watson Research Center, n.pag., 1995.

[VN96] VanHilst, M., and Notkin, D., “Using role components to implement collaboration based design”, OOPSLA: The First Eleven Years Conference Proceedings 1986-1996. CD-ROM. ACM Press, New York, 1996.

[WB93] Wirfs-Brock, R., “Stereotyping: A technique for characterizing objects and their interactions”, Object Magazine, 3(4), Nov/Dec 1993.

[Webs95] Webster, B., Pitfalls of Object-Oriented Development, MIS Press, New York, ISBN 1-55851-397-3, 1995.

[Weis97] Weisart, C., “Point-Extension Pattern for Dimensional Numberic Classes”, ACM SIGPLAN Notices, 32(11), pp. 17-20, Nov. 1997.

[WF86] Winograd T., and Flores, F., Understanding Computers and Cognition: A new Foundation for Design, Ablex Publishing Corporation, Norwood, New Jersey, 1986.

[WF97] Winant B., and Frankel, M., “Solving object state model mysteries using a key event dictionary”, JOOP, 10(1), pp 52-58, Mar-Apr. 1997.

[WfMC95] Workflow Management Coalition Working Group 1A, Workflow Process Definition Read/Write Interface: Request For Comment, Document Number WFMC-WG01-1000, February 17, 1995, Workflow Management Coalition, Brussels, Belgium, 1994, revised 1997. (ftp://ftp.aiai.ed.ac.uk/pub/projects/WfMC/if1/wg1_1000.pdf on 11/16/97)

[Wieg97] Wiegers, K., “Use Cases: Listening to the Customer's Voice”, Software Development, March 1997, Vol 5., No.3.

[WL97] White, S., and Lemus, C., “Architectural Reuse through a Domain Specific Language Generator”, Eighth Annual Workshop on Institutionalizing Software Reuse (WISR), 3/23/97, Ohio State University, March 23-26, 1997. (http://www.umcs.maine.edu/~ftp/wisr/wisr8/papers/whiteS/whiteS.html on 7/23/97)

[Wood95] Wooding, T., “The Feasibility of Common Business Objects”, Object Magazine, 5(6), Oct. 1995. (http://www.compulink.co.uk/~oig/cbo.html 5/10/96)

[WR92] Whittle, B., and Ratcliffe, M., “Software component interface description for reuse”, Software Engineering Journal, 8(6), pp. 307-18, Nov. 1993.

[WS93] Wu, W., and Sakauchi, M., “A multipurpose drawing understanding system with flexible object-oriented framework”, IEEE Proceedings of the Second International Conference on Document Analysis and Recognition, 10/20/93, Tsukuba Science City, Japan, Comput. Soc. Press, Los Alamitos, CA, ISBN: 0 8186 4960 7, pp. 870-3, 1993.

[Your93] Yourdon, E., Decline and Fall of the American Programmer, Yourdan Press, Englewood Cliffs, New Jersey, ISBN0-13-191958-X, 1993.

[Your94] Yourdon, E., Object Oriented Systen Design, Yourdan Press, Englewood Cliffs, New Jersey, ISBN0-13-626325-3, 1994.

[Zave95] Zave, P., “Classification of research efforts in requirements engineering”, Second IEEE International Symposium on Requirements Engineering, 11/20/95, 1995. (http://www.research.att.com:80/orgs/ssr/people/pamela/re.ps.gz on 7/15/96)

[Zeie95] Zeien, G, “Weaving a web of use cases”, Object Magazine Nov-Dec. 1995.

[Zorm95] Zorman, L., “The context and composition of scenarios”, OOPSLA workshop – Requirements Engineering: Use Cases and More, 10/15/95, Austin, Texas, 1997. (http://www.isi.edu/soar/lorna/oopsla95.ps on 2/5/97)