

Managing Domain Architecture Evolution Through Adaptive Use Case and Business Rule Models

Russell R. Hurlbut
Expertech, Ltd.
P.O. Box 4151 Wheaton, IL 60189
Document: XPT-TR-97-02
rhurlbut@charlie.iit.edu

ABSTRACT. *Among object oriented analysis and design practitioners, use cases have become the preferred method for capturing functional requirements of applications. This is evidenced by its inclusion in the Object Management Group's Unified Modeling Language (UML) specification. Although the UML version 1.1 meta-model includes detailed mappings of use cases to other modeling constructs that implement these use cases, the elaboration of use cases for requirements gathering and domain analysis is ambiguous. The Workflow Management Coalition's Workflow Reference Model (WRM) issue 1.1 provides considerably more detail in its definition of a workflow, but in a less formal meta-model. This paper formalizes a use case centric approach to facilitate management of the evolution of domain architectures that is compatible with both models. This is accomplished through the integration of three constructs: adaptive frames, speech acts and business rules. This approach creates a set of use case formalisms that provide semantic guidance in the construction of higher level use cases tailored specifically for participants in domain analysis activities rather than architectural design. Through adaptive frames, use cases are evaluated from a larger context that considers the evolution of the supporting domain model. Through speech acts, the context for use cases is expanded in second dimension that considers the dialog between an actor and system as part of a larger business process. Through business rule bindings, use case action sequence templates are abstracted from the specific details of any application instance.*

KEY WORDS: *Use Case Formalisms, Adaptive Frames, Speech Acts, Domain Architecture, Workflow, Business Process Modeling, Business Rules, Object Oriented Analysis and Design, Unified Modeling Language, Workflow Reference Model, Dynamic Objects*

1. Introduction

The Object Management Group's Unified Modeling Language (UML) specification version 1.1 suggests several different formats for representing a use case [UML97]. Although the most common form for representing a use case has been in plain text, the UML Semantics guide also identifies alternative use case descriptions in the form of operations, activity diagrams, and state-machines. Other behavior description techniques, such as pre-and post conditions, are also permitted.

The Workflow Management Coalition's Workflow Reference Model (WRM) issue 1.1 provides a definition of a business process that is very similar to the UML description of a use case [Holl94] [WFMC96]. They do not specify the format for the process definition, simply stating that it can be expressed in textual, graphical, or formal language notation. However, their meta-model does provide some structure by describing discrete activity steps, rules governing the progression through the activity steps, and data that is used to determine the transitions.

Plain text representations of use cases alone can vary widely in their presentation. Examples of textual formats include free form narratives, tabular, structured form templates, formal language

expressions, and scripts. In addition to the graphical representations for interactions and state (for representing activity diagrams and state-machines), use case structures, rules, and implementations can be represented graphically. Furthermore dynamic forms of use cases such as visualizations, storyboards, and role-playing are possible. The wide range of possible use case formats allows one to tailor the presentation to the appropriate target audience based on their purpose and applicability to various use case levels and life-cycle stage [Hurl97d].

Frame technology is a software reuse technique built around the definition of archetypes and deltas [Bass97]. This runs contrary to mainstream object-oriented implementation languages, which require strict inheritance of attributes and operations from ancestors in its class hierarchy. Rather than focusing on similarities, adaptive frames are constructed by identifying the differences. One side effect of this shift of emphasis is the fusing of the concepts of inheritance and aggregation into a single mechanism. Since the UML is essentially silent on how to provide structural representation of use cases, adaptive frame technology offers guidance in forming a coherent approach for structurally organizing use cases. Consistent with UML semantics, use cases are viewed as collection of prototypical course of actions. The archetype is designated as the basic course of action. The deltas, or alternative courses of action, can then be easily incorporated to evolve the use case as the underlying domain model and architecture matures. In this way, additional functionality, exception handling, and refined courses of actions can be accommodated.

Speech acts have been used for over a decade to describe business process workflows [WF86]. Request, accept, declare, withdraw, and assert are all examples of speech acts that are relevant to use case modeling. Speech acts can take the form of messages to more formally structure the interactions between an actor and the system being modeled by use cases. The ActionWorkflow model consists of a workflow loop to represent a dialog between a customer and performer [MWFF92]. For use case modeling purposes, the customer maps to an actor and the performer maps to the system being modeled. This workflow process consists of four phases: preparation, negotiation, performance, and acceptance. In most use case courses of action, one or more of these phases will be implied. However, this model provides an excellent basis for delineating between what constitutes a new use case and what is merely a variant of an existing use case.

Business Rules formalize an approach for identifying and articulating the rules that define the structure and control the operation of an enterprise. The UML and WRM provides the ability to document business rules to a degree, many of the constraints under which the enterprise operates have not been articulated as well, if at all. The GUIDE Business Rules Project was organized to address this situation [HH95]. This project defined and described business rules and associated concepts. They have also created a meta-model of business rule constructs. One of the contributions is the identification of business rules into one of four categories: definition of a business term, facts relating terms to each other, action assertions in the form of constraints, and derivations via mathematical calculations or inference. The UML makes clear distinctions between structural and behavioral models, but policy models do not exist. Unlike the behavioral models of the UML and the workflow definition model of the WRM, business rules do not describe the steps to be taken to achieve the transition from one state to another, or the steps to be taken to prohibit a transition. A business rule is declarative rather than procedural in its nature. The GUIDE project deferred for future work the goal of providing a rigorous basis for engineering new systems based on formal definitions or reverse engineering business rules from existing systems.

2. Definitions

This section sets the context for Adaptive Use Cases and Business Rule Models introduced in this paper. We describe the models introduced in the first section in greater detail. Specifically we

present 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, in turn, we present each of the conceptual elements that form the foundation for this approach as then discuss how they relate to the UML specification. These components are the Workflow Application Programming Interface 1 from the Workflow Management Coalition's Workflow Reference Model, speech act semantics based on the ActionWorkflow model, adaptive modeling elements based on Basset's frame technology, and business rule constructs and mechanisms.

2.1. UML 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 defining a use case, two additional terms become emerge in importance. 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 constraint the UML places on potential use case formalisms can may be defined through its extension mechanisms.

2.1.1. Use case

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

- *Glossary*: “The specification of a sequence of actions, including variants, that a system (or other entity) can perform.”
- *Notation Guide*: “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.”
- *Abstract Syntax General Presentation*: “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.”
- *Abstract Syntax Metaclass Definition*: “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**.”
- *Dynamic Semantics*: “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.”

2.1.2. 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*” and a use case instances 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.”

2.1.3. 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.

2.1.4. Discussion

The inclusion of the «extends» and «uses» have been inherited from the Jacobson, et. al. OOSE (Object-Oriented Software Engineering) 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.

The essence of the difficulty with «extends»¹ and «uses» lies in the definition 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

¹ Names of stereotypes are delimited by guillemets and begin with lowercase (e.g., «type»).

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.

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], then a “using” use case must consider all of the scenarios of the “used” use case. Here, the ‘complete usage’ takes on additional meaning. From one perspective, “complete usage” represents a series of interactions with the system constituting its intended usage, e.g. until the actor has attained a desired goal. From another perspective, ‘complete usage’ represents the full range of possible alternative behaviors at any 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. 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. The complexity of the «uses» generalization is not specified at the same level of abstraction that the relationship is defined. Rather, it is an aggregation of the nested scenario relationships contained within the two use cases.

A use case modeler needs to be careful with how the UML model elements are applied. The following points summarize suggestions 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 «use» 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. Instead group together scenarios that can be adequately expressed in a single sequence diagram. 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. This runs contrary to the standardization of the UML notation and semantics. The comparison between use cases and classes as subtypes of a classifier breaks down. The use case as a stereotyped classifier should be tightened.

2.2. WRM Workflow Definitions

The Workflow Management Coalition's Workflow Reference Model (WRM) issue 1.1 provides several definitions related to a business process:

- *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.

The workflow process definition read/write interface provides a formal specification to access workflow definitions [WFMC95]. Several standard entity classes and attributes are defined. These are shown in Figure 2-1 using UML notation with some allowances for naming conventions. These structures are part of the Workflow Application Programming Interfaces Interface 1 (WAPI-1) portion of the Workflow Management Coalition's Workflow Reference Model. Interface 1 describes three levels of compliance: A, B, and C. Level A 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. *OpenNewAccount*. The intent of the WAPI-1 interface is to enable queries about use case instance, such as *OpenNewAccount*, with respect to the current state of the workflow process and subordinate activities. 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 of the various representations for a use case that are desired to be used should be mapped to a specific Format definition shown in Figure 2-1. 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 no formal represented for structured text representations.

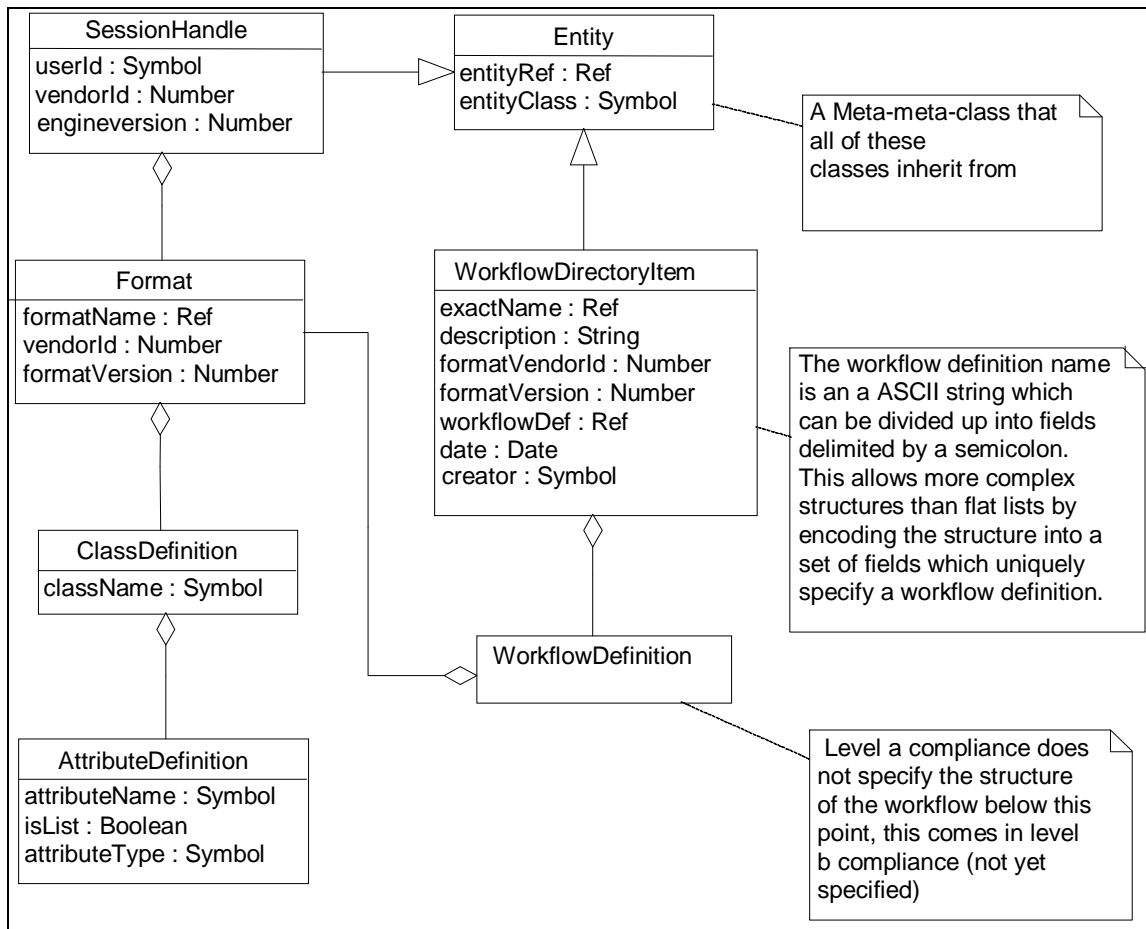


Figure 2-1: WRM Interface 1 Specification in UML notation format

Our approach for accommodating the WAPI-1 specification by using UML modeling constructs requires use its extension mechanisms. Once mapping to structured text use cases is defined through the UML extensions, mappings to use case refinements and realizations are possible. 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 [Hurl97g]. This reification can be found as a recurring architectural pattern in several applications and development environments [Beed97].

2.3. Speech Acts

Integrating the concept of speech acts into use case modeling places additional constraints upon use cases as defined in the UML. 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. Such a complete usage is also referred to a ‘a coherent unit of functionality’.

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.

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 following use case example is used to illustrate how speech acts are used to add additional constraint to use cases with respect to system boundary. This example is provided in narrative format. A structure version will be presented later:

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 (ChecksStatus)

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.

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. The system boundary may be 1) automated portion of the student registration system, 2) the entire university computer system including e-mail, or 3) the full information system incorporating both automated and manual processing.

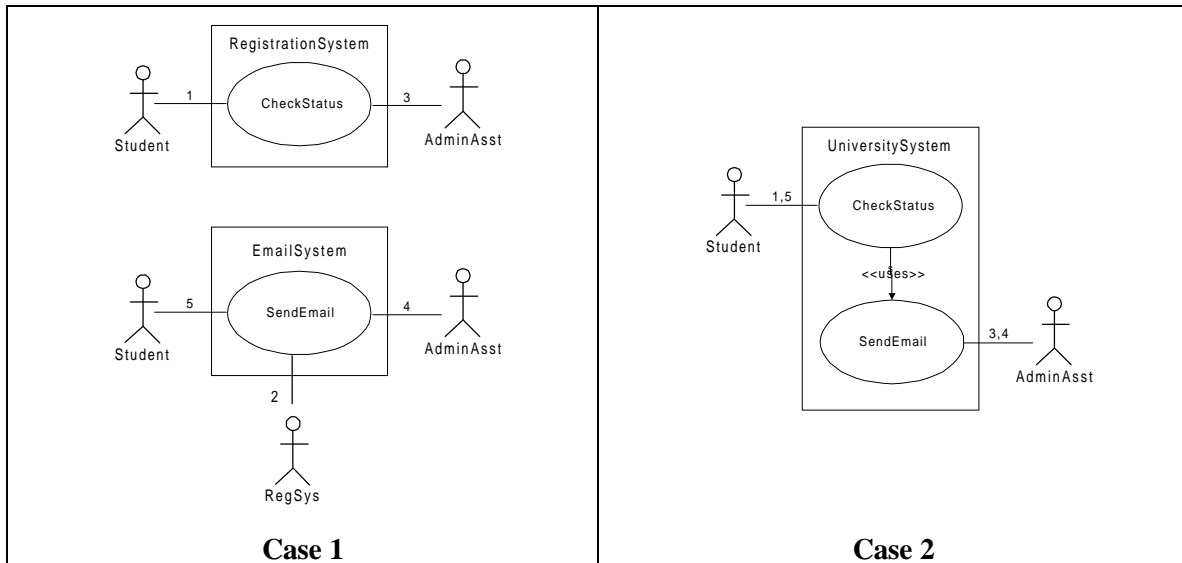


Figure 2-2: Use Case System Boundaries Cases 1 and 2

In the first system boundary case (Figure 2-2, Case 1), actual completion of the workflow may occur as the result of interaction with two actors, the student and administrative assistant. Two systems must be modeled to provide an accurate depiction of the workflow, the registration system and the e-mail system. Consequently, there is no continuity of workflow between the two actors. Although the student has no interest in how the department administrative assistant is notified, these details are modeled as part of the use case. In fact, the actor is the email system and not the department administrative assistant. When the system initiates a message to the administrative assistant via the email system, there is no observable result of value to the student. The inclusion of the email system as an actor primarily serves to highlight the system boundary.

The second system boundary case (Figure 2-3, Case 2) includes the email system as part of a larger university system. It also identifies the department administrative assistant 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 does expose 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 2-3 illustrates both the basic use case and 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.

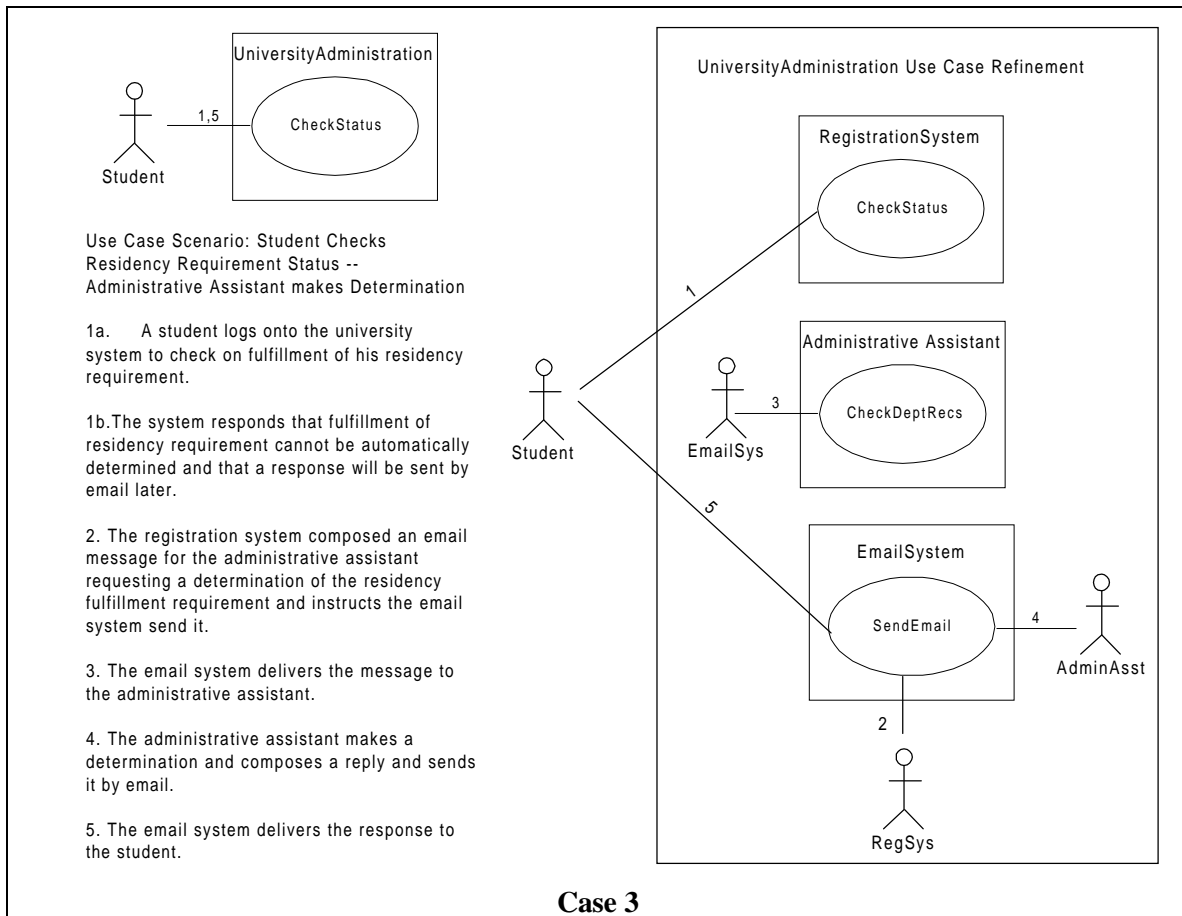


Figure 2-3: Use Case System Boundaries Case 3

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 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, and as was demonstrated in Figure 2-3, use case refinement changes the scope and identity of the actor-system dialogs that comprise each use case action segment. This leads to a recursive refinement of use cases as shown in Figure 2-4.

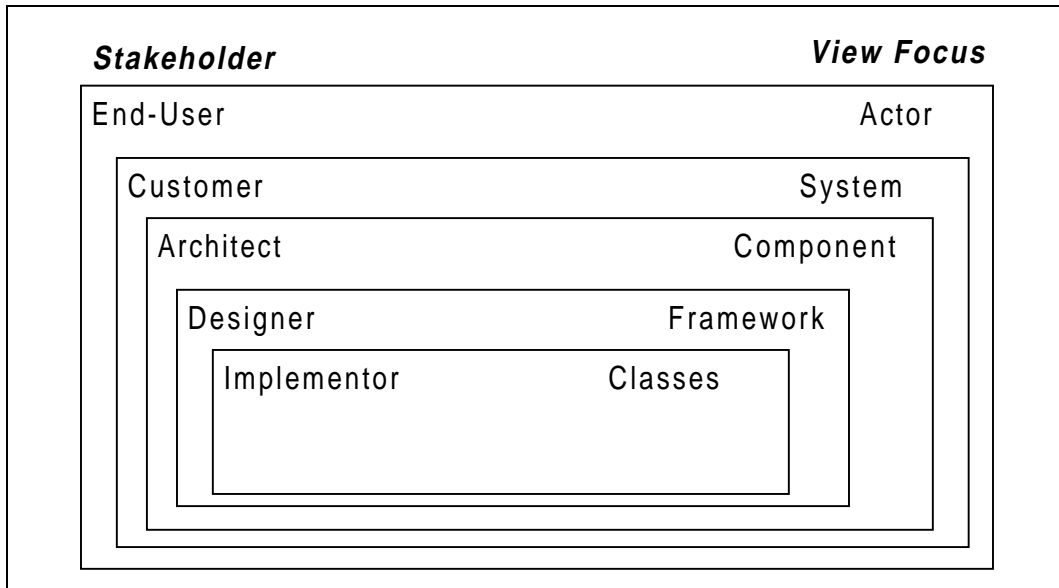


Figure 2-4: Use Case Refinement Levels

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 directly 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, allowing a glass-box view that selects and reveals an appropriate level of refinement.

Use cases were originally intended to depict how humans enter and interact with OO systems. 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, we have several scenarios of interaction as shown in Table 2-1.

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

Table 2-1: Customer-Performer Interactions between Human and System

2.4. 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 focused on the system boundary and secondary actors, integration of adaptive frames focuses on 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 and 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.

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. Appendix A provides a complete description of meta-attributes for both `UseCase` and `Class`.

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 `Place-Order`, `Check-Status`, and `Establish-Credit`. 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 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.

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 provide allow management of finer specification detail.* Scenarios keep the use case as a specification model rather than implying realization through 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 *Class* 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 much greater opportunities for use case reuse than the large grained generalization stereotypes defined in UML.

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 base line 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.

2.5. Business Rules

The approach of integrating business rules with use cases in this paper introduces a policy-oriented view to domain models. Workflow oriented rules are primarily used for process flow control. By introducing business rule bindings to use cases, the static object and data structures of an application can also be controlled. However, unlike class model which directly integrate 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 templates to create sets of scenarios that comprise a 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.

We believe that the binding of business rules to use cases represents a novel approach that sets it apart from previous approaches. As evidenced by the approaches that are described in this section, there has been considerable discussion with respect to business rules that *suggest* a relationship

between use cases and business rules. To our knowledge and comprehensive research regarding use cases, the approach describe in this paper is the first to formalize the relationship.

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]. 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 as they relate to processes that transcends 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. 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 approach here is similar to the dynamic reflection approach.

The presentation of our parameterized business rule binding approach, which will be detailed in section 4, 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 tracks 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 assumes a certain architecture and 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 the 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's nested rule model that provides a multi-level representation of the declarative and active behavior through rule sets, coupling modes and interactions has also been proposed [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 typically relate one attribute to another, triggers relate operations to attributes, and exception handling which can be any combination. However, since all 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. Our approach of binding business rules to process models provides a high 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.

3. Adaptive Use Cases

3.1. UML Extension Mechanisms

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 metaclasses with new attributes and constraints. A new graphical representation may also be introduced 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 "pseudoattributes" of the stereotype to supplement the real attributes supplied by the base element class. The values permitted to such "pseudoattributes" 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.

3.2. Related Use Case Model Constructs

3.2.1. 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. Modeling Actors in this manner simplifies the use case model by abstracting the similarities of interactions between use cases and the Actor. 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 Actor subtype 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

3.2.2. UseCase Template

A UseCase template is a UseCase that contains one or more template parameters. In this 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.

Variation Aspect	Mechanism	<i>Renew Library Item Example</i>
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 3-1: Actor Influences

Variation Aspect	Mechanism	<i>Renew Library Item Example</i>
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

Table 3-2: System Influences

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. This approach 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 3-1 represent decision made by the actor which primarily determine the actual scenario that is performed. The system influences shown in Table 3-2 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.

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 simultaneously 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. Our 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, so that only the details that are under consideration can 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 that an actor subtype can participate in, 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 use case extensions are strengthened by its affinity with existing UML semantics.

3.2.3. 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 that 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 would be. For example, using the `renew library item` from a previous example, 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.

3.3. 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 in 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 use case model.

3.4. UML Extension for Managing Domain Architecture Evolution

This section provides the definition of each new stereotype, taggedValue and constraint needed to model the Adaptive Use Case mechanism. The stereotypes described in this section overcome these ambiguities with respect to extension points and variability mechanisms in the UML specification.

3.4.1. Model Elements

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. Any constraints are also noted. Figures 3-1, 3-2 and 3-3 illustrate the extensions. Each stereotype may be found in one or more of these figures.

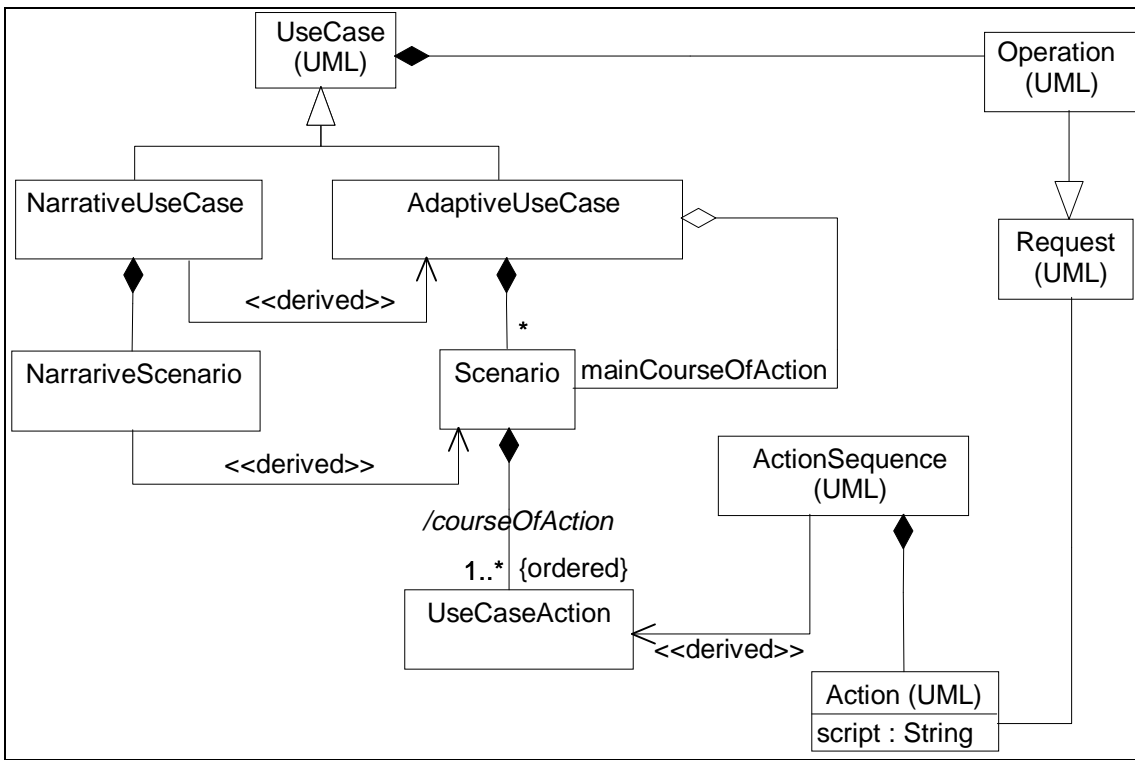


Figure 3-1: Adaptive Use Case Stereotypes

NarrativeUseCase: a 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 meta-attribute, 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 *narrativeDescription* of type `String`.

NarrativeScenario: a scenario that contains a narrative description of a sequence of actions. It is derived from *Scenario*. The mapping for this derivation is 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. It is generally used to fully describe an instance of *Scenario*.

AdaptiveUseCase: a 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 a *AdaptiveUseCase* is the set of all interfaces for the use case with a single actor.

Scenario: 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.

ArchetypeScenario: a specialization of *Scenario*. It contains an ordered sequence of actions through its list of *UseCaseSegments*. *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*.

VariantScenario: 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.

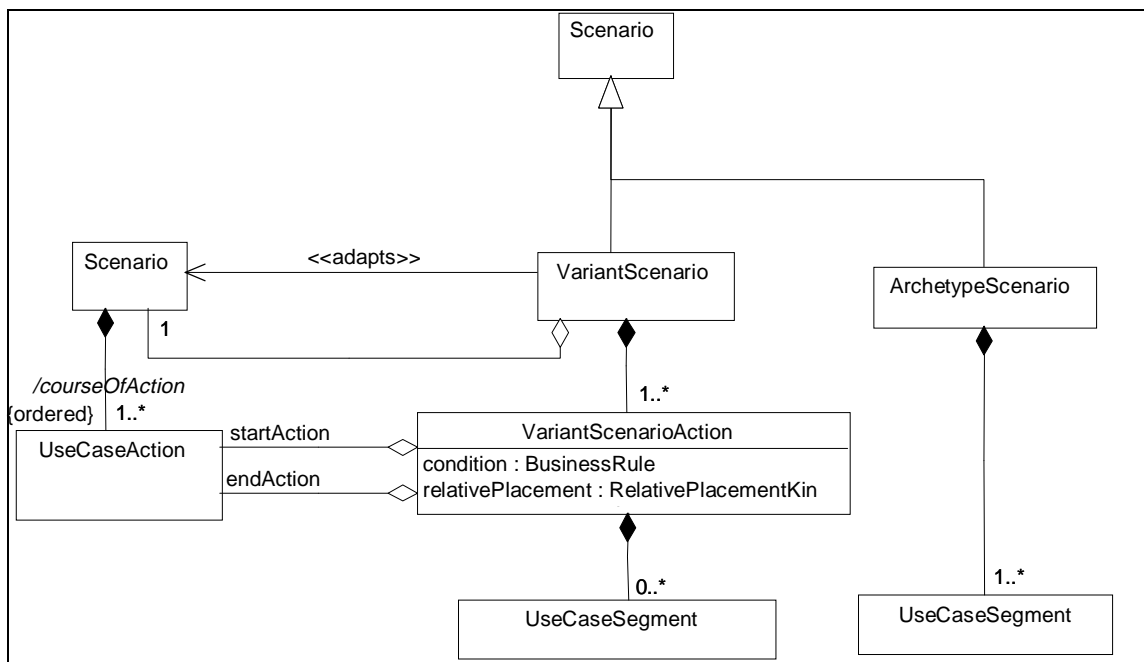


Figure 3-2: Scenario Stereotypes

VariantScenarioAction: 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.

«*adapts*»: 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.

BusinessRule: an ordered set of derivations, structural assertions, and action 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 action 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.

RelativePlacementKind: a stereotyped Enumeration that defines how the actions associated with a *VariantScenarioAction* will be inserted into the base scenario identified by a *VariantScenario*. Only *replace*, *extend* and *append* are base placement kinds. The others are derivatives of these three. However, since the underlying base scenario may evolve, the derived placement kinds convey additional semantic meaning that preserves intentions that may otherwise result in unexpected behavior. For example, creating an *extend* on the first action is semantically identical to creating a *preface* as long as the underlying base scenario first action remains unchanged. If a new action is inserted into the scenario as the first action, a semantic difference will occur. With the *extend* usage, the *VariantScenarioAction* will precede the same action as before, but now there is another action that will occur before it. With the *preface* usage, the *VariantScenarioAction* will always be performed first (provided the condition expression evaluates to true. The *globalExtend* and *rangeExtend* placement kinds provide a shorthand description for actions that may occur at and time. For example, for a ‘dining at a restaurant’ use case, a variant scenario may allow the actor to perform the action ‘visit the restroom’ at any time. This can be expressed by a *globalExtend* once, without the need to associate this possibility with each action. If additional actions are added to the base scenario, they will automatically allow this conditional action to occur. A *rangeExtend* restricts the set of actions that the *VariantScenarioAction* may intervene. The actor in the ‘dining at a restaurant’ may only order drinks after the ‘get seated at table action’ occurs. The complete set of values for *RelativePlacementKind* are:

- | | |
|----------------|---|
| <i>Replace</i> | 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. |
| <i>Extend</i> | New actions are attached before any the starting action. The ending action is ignored. This is identical to a <i>replace</i> 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. *Extend* is preferred, since the intention is much clearer.

- Preface* 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 extend.
- 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 any the starting action. The action sequencing resumes prior to the starting action. In other words, the ending action must precede the starting action.
- Loop* New actions are attached before any the starting action. The action sequencing permits reevaluation of the condition. The ending action is ignored. This is a special case of *regress*.
- RangeExtend* New actions are attached before *each* action in range. The condition is evaluated each time. This represents a special case of *extend* that provides a concise representation rather than enumerating each one. It also will accommodate adjustments to changes in the base scenario, such as actions added or removed within the range.
- GlobalExtend* New actions are attached before all actions. The condition is evaluated each time. Starting and ending actions are ignored. This represents a special case of *rangeExtend*.

UseCaseAction: 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.

UseCaseSegment: 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 as described by Gamma, *et. al.* [GHJV85].

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

UseCaseTemplate: a use case 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.

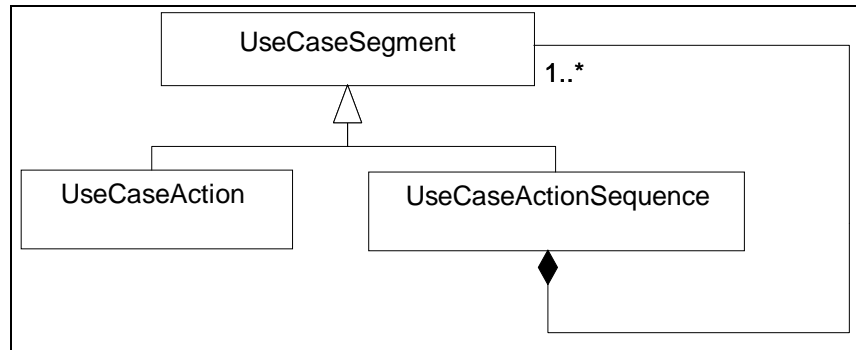


Figure 3-2: UseCaseSegment Stereotypes

3.4.2. Well-Formedness Rules

The UML Abstract syntax for a Use Case provides for a single attribute and five well-formedness rules. This section extends the semantics of a use case by providing associated well-formedness rules for the stereotype described in the previous section

- A UseCase can have only one association with an Actor for a given level of abstraction. Secondary actors may only be represented through refinements to the use case.
- An actor must know what use case is invoked at its commencement (but not necessarily which scenario).
- A Use case must contain an ordered set of actions that comprise a complete speech act dialog that conforms to the Winograd/Flores basic conversation for action model.
- Alternative sequences of actions not perceived from the outside should be suppressed.
- A *AdaptiveUseCase* must contain exactly one *Scenario* that is designated as the *mainCourseOfAction*
- An *ArchetypeScenario* does not contain any conditional branching.
- An «adapts» stereotyped dependency relationship between a *VariantScenario* as the source and another *Scenario* as the target must contain at least one *VariantScenarioAction* element.
- The *RelativePlacementKind* selected restricts the starting and ending action associations that can be designated.
- An *ArchetypeScenario* may play the role of a *VariantScenario* in an «adapts» stereotyped dependency with another *ArchetypeScenario* provided the target *ArchetypeScenario* is in a different Package.

The static semantics of UML define how an instance of a construct should be connected to other instances to be meaningful. The Object Constraint Language (OCL) is used to express these static semantics. The constraints take the form on invariants that must be satisfied for the construct to be meaningful. Additional operations may also be needed for the OCL expressions. The complete OCL well-formedness rules for the adaptive use case extension stereotypes will be detailed in [Hurl98a].

3.5. Discussion

The UML Semantics documents states that each method performed by a use-case instance is performed as an atomic transaction. This means that it can not interrupted by any other use case instance. For traceability purposes, it is recommended that a one to one mapping between

operations and methods should occur. Furthermore, each operation should be identified as an extension point.

The adaptive use case mechanism provides for this direct mapping as shown by the derived mapping between a use case action and the UML ActionSequence model element. 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.

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 action assertions. Furthermore, many of these may be derived from the formalized representations in adaptive use cases.

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. The UML prescribes strict inheritance in this context in order to encourage reuse of implementation rather than preserving behavior. The rationale behind this policy is that most implementation environments utilize strict inheritance (i.e. features can be replaced or added, but not deleted). The inheritance policy follows this line by disabling refinements that may lead to non-strict inheritance once the state machine is implemented. This policy restricts the degrees of freedom for variability mechanisms. 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.

The adaptive use case mechanism is intended allow dynamic modifications to use case scenarios that are contained within the same model management package. Jacobson elaborates on the packaging constructs with several additional package stereotypes [JGJ97]. The «configuration item» and «configuration» stereotypes are used to represent a configuration item(s) grouped into a configuration. For example, one «configuration item» could be an actor and use case that has a variation point. A second «configuration item» would define the variant behavior that is plugged into variation point. The «configuration» stereotype package would then contains these two configuration items. These stereotypes are compatible with the adaptive use case stereotypes defined here. Since both imported scenarios and use case segments can only be referenced through their derived lists of use case actions, reuse across package boundaries is always a black-box representation. An internal reorganization within the imported package may occur without any impact on the client package.

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 referred to here is really a meta-domain, representing business domain architecture evolution management.

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 the following section.

4. Business Rule Models

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. This separation of policy, process, and structure provides for greater flexibility and more concise and understandable representations. In this section we will develop a business rule model that is compatible with the model elements of the GUIDE Project. As will be shown, this business rule model integrates with our adaptive use case model. We will also detail how the various types of business rules are used to construct the business rule parameter templates and then how they are subsequently bound to adaptive use cases.

4.1 Integrating Adaptive Use Case and Business Rule Model Elements

In order to integrate the use case action statements from an adaptive use case model with rules from a business rule model, some structure and formalism must be imposed on the action statements. Use case actions provide the mechanism that links the process oriented adaptive use case model to structure oriented collaboration models. Likewise, use case actions also link the procedural order on actions contained in the adaptive use case model to **non-procedural business rule models**.

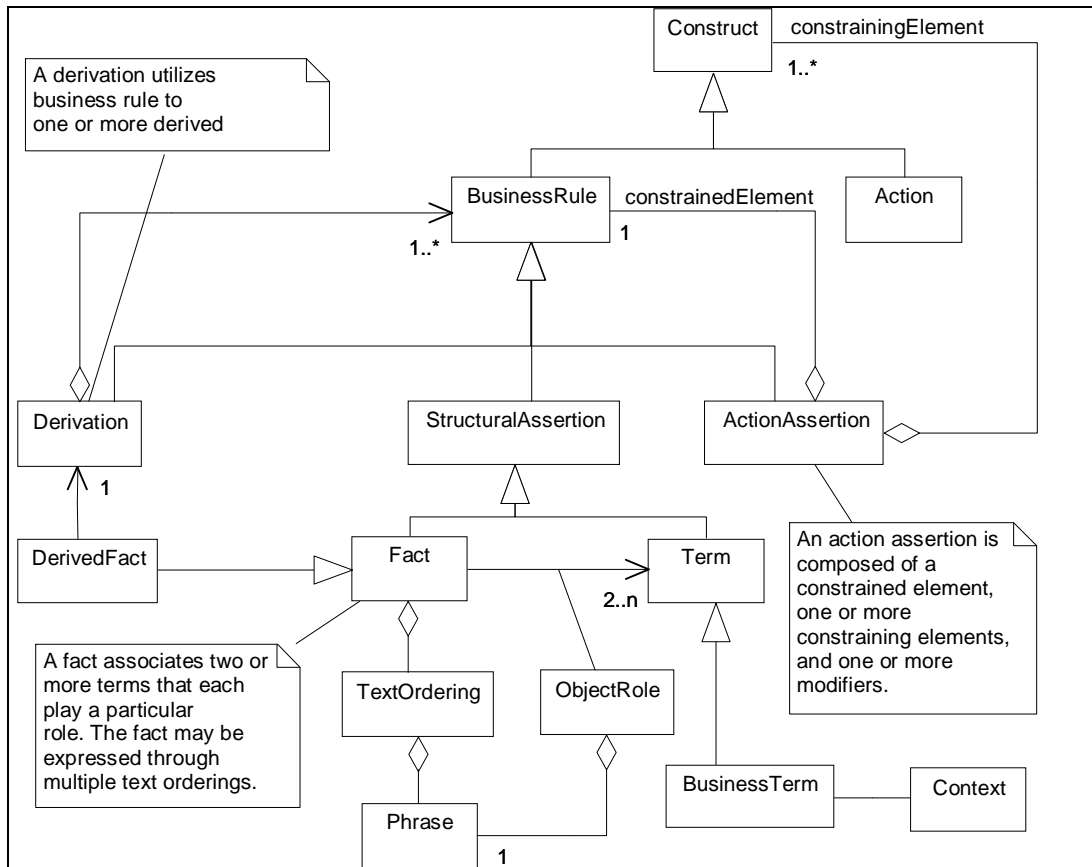


Figure 4-1: GUIDE Business Rule Model Constructs and Contexts

The GUIDE Business rule model contains two model elements that permit linkage of an adaptive use case model and an adaptive business model. These are the *Action* model element and the *Context* model element. The *Action* model element maps directly to the *UseCaseAction* contained in the Adaptive Use Case meta-model extension to the UML specification presented in the previous section. The *Context* model element maps directly to the association between an *Actor* and *UseCase*. As shown in Figure 4-1, both *Action* and *BusinessRule* are *Constructs*, sharing equal status as first class elements in the business rule model.

Where the adaptive use cases specify procedural order on actions, business rules **specify constraints on the results that actions can produce. The constraints are represented as non-procedural business rule. Based on the GUIDE business model semantics, these business rules can be derivations, structural assertions, or action assertions. Derivations are classified as either mathematical calculations or inference. Structural assertions are either business terms or facts relating terms to each other. The structural assertions are primarily used to represent facts that describe possibilities in the domain model. Derivations merely add additional facts. Actions bring life to these facts, but only after conforming to constraints imposed on them by the action assertions. The action assertion is composed of a constrained element, constraining elements, and optional modifiers. Using Ross's nomenclature, the constrained element and constraining elements can also be referred to as the rule's anchor and correspondents, respectively. The anchor is the type that the rule belongs to (i.e. the rule is dependent upon the anchor in a manner similar to third normal form data normalization). Correspondents are other types that are needed in order to evaluate the rule. Modifiers related to the anchor are referred to as interpreters and modifiers related to the correspondents are referred to as qualifiers [Ross97].**

4.1.1 Action Assertion Extensions

Under the GUIDE business rule model, action assertions are classified in three different aspects. The semantics of the eighteen permutations of the classification are not well defined. In this section we will refine the definitions and provide several model element extensions.

4.1.1.1 Computation Types

The first aspect classifies an action assertion as an *Enabler*, *Timer*, or *Executive*. This classification is open for extension. Based on the nature of the constraining object, the *Enabler* permits the creation of a new structural assertion instance, enables another action assertion, or permits the execution of an action, and the *Timer* tests or sets a structural assertion, or enables another action assertion. An *Executive* causes the execution of one or more actions.

For purposes of our integrated business rule model, we are primarily interested in three facets that are associated with an action assertion. First, it possesses a truth-value. This corresponds with an *Enabler*. Second, it may also possess a result, such as a derived fact, a count, or a time interval. In the case of a time interval, this corresponds with a value yielded from a *Timer*. Third, it may possess the ability to cause execution of an action. This corresponds with an *Executive*. **The three facets can all be revealed in a business rule, such as:**

```
IF the library item is more than three weeks overdue  
THEN revoke the lender's lending privileges
```

As shown in this example, three action assertions would be required. The *Timer* (three weeks overdue) enables the condition (the IF clause), which is an *Enabler*. The *Enabler* enables the integrity constraint (the THEN clause), which is an *Executive*. The *Executive* invokes the action.

Rather than deal with the fine grained GUIDE classification types, we will label such composites under the heading of business rule patterns and only refer to these larger constructs. In this manner we conform to the GUIDE model, while at the same time provide derived extensions that expedite integration with the use case model. The example provided will be referred to as the *Contingency* business rule pattern. Table 4-1 provides a mapping of business rule patterns (BRPs) to five other rule modeling approaches referenced in this paper.

Patterns/Approach →	UML	GUIDE	A/OOBTMT	Ross	SOMA	Comments
Attribute						
Range	★	★	✓	✓	✓	e.g. age between 21 and 65
Interval	★	★	✓	✓	★	increment-by, rounding
Domain	✓	★	✓	✓	✓	for enumeration types
Type checking	★					include conversion between types
Default	✓	★		✓	★	initial value if null
Collection						
Multiplicity	✓		✓	✓	★	mandatory, limits, and cardinality
Ordering	✓			✓	★	key function definition
Unique	★	★	✓	✓	★	relates to values, not object identity
Recursive	★		★	✓		e.g. composite pattern
Meta-attribute	✓		★	✓		e.g. frequency distribution of values
Logical Connections						
Sub-typing	✓	✓	★	✓	✓	includes disjunction and overlapping
Association	✓	✓	✓	✓	✓	and, or, if exists, for all
Temporal						
Association	★	★	★	✓		relates creation to others' existence
Historical	★	★	★	✓		functional versioning of attribute
Meta-attribute	★			✓		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
Timed Execution	★	✓	★	✓		absolute, relative executions
Routing		★		★		role, organization structure mapping
Authorization		✓	★	★	★	creation, override, execution rights
Contingency	★	★	✓	✓		timed window for execution
Enforcement		✓		✓	★	prevent, notify, log, permit
Self-Modify						reconfigure, recompile components
Derivation						
Algorithm	✓	✓		✓		mathematical calculations
Production		✓		✓		inferencing, induction, deduction
Protocol						
Contract	✓	★	★			arguments, types, order
Alternatives	★	★				alternative, extension actions
Exceptions	★	★	★	★	✓	conditions for exception actions
Fire Sequence		★	★	✓		operation, rule invocations
<p>Legend: ✓ = explicit or strong support ★ = implicit or sufficient support = unintuitive support – either requires compound constructs to support, or is unsupported</p>						

Table 4-1: Comparison of Business Rule Patterns to Other Rule Construct Approaches

4.1.1.2 Expression Types

The second aspect of action assertions concerns how the anchor and correspondent are expressed. One of the three formats shown below may be used, where the anchor is represented by an 'x' and the correspondent by a 'y':

```
authorization:           [only] x may do y
condition:               if x [then y]
integrity constraint:    x must [be|have|do] y
```

Integrity constraints enforce, but conditions only indicate a truth-value (i.e. a true/false Boolean). Consequently, a condition can not directly cause the execution of an action, it can only enable a structural assertion (i.e. create an instance of a fact) or another action assertion that can execute the action, typically an integrity constraint. A condition that enables another assertion is represented in an if...then... format.

From the perspective of the business rule model semantics, an *Action* cannot be constrained, only persistent instances of entities can be constrained. In other words, we can constrain the data left behind after execution of an action, but not the action itself. The action is considered an atomic block-box process that is viewed as a transaction.

Although the GUIDE model prohibits actions from being rule anchors, actions can still be associated with a rule anchor. We introduce a new action assertion expression to address this type and promote its importance. These types of rules represent control over workflows that cross the system-human boundary as described in Table 2-1. Our *WhenNeeded* business rule pattern is a composite of a condition that is tied to an action as a pre-condition and an executive action assertion. This action must be an access to retrieve the value for an attribute. It takes this form:

```
IF the value for attribute x is requested
AND it is not known
AND it cannot be derived from the rule set
THEN execute action y.
```

4.1.1.2 Modifier Types

The third aspect of the action assertion represents a modifier. Only one is specified as a built-in element of the GUIDE business rule model. However, the business rule modeler has implied freedom to also extend the types of modifier, since they relate directly to new types of action assertion that can be declared. The built-in modifier changes a controlling assertion to an influencing assertion (i.e. *must* versus *should*). We have created a set of modifiers that subsumes the GUIDE modifier and Ross's qualifiers and rule interpreters [Ross97]. They are shown in Table 4-2. Rather than treat these modifiers as first class constructs in the business rule model as GUIDE and the Ross Method prescribe, we choose to treat these as attributes of rules that can be used in business rule cards when specifying a rule. Business rule cards are form templates to aid in the construction of business rules. These should not be confused with business rule templates, which are parameterized business rules. Instead they are fill-in-the-blank worksheets, not modeling constructs. They are intended to help structure the narrative of a rule and create appropriate associations with other rules or actions.

Modifier	Description
State	Essentially, the state of the rule is either unknown, true, or false. However, in order to handle special circumstances, three Boolean attributes are defined for each rule – isTrue, isEnabled, and isInvoked. If not Enabled, then other two values are unknown. If invoked, then a truth-value other than unknown implies a successful invocation. Otherwise, the rule firing caused an exception.
Inverter	Inverts the usual meaning of a term, such as changing <i>least</i> to <i>most</i> .
Discriminator	Uses the rule's anchor as a discriminator to identify the constraining element instances (I.e. selection criteria)
Accessor	Provides selective references to CRUD events. Can represent single or multiple combinations.
Initiator	Controls when a rule will fire, such as <i>immediately</i> or by reference to a state and/or time.
Enforcer	Extends the enforcement semantics for the built-in controlling versus influencing modifiers to include require (controlling), pardon (permit an exception, but prohibit future violations), notify (influencing), and log (always permit, but log the test).
Power type	Applies the rule to an instance of the meta-class rather than the instance itself.
Logical Quantifier	Selects elements based on the following logical operators: \exists (there exists), \forall (for all)
Logical Connector	Selects elements based on the following logical operators: \wedge (and), \vee (or), and \neg (not).
Enumerator	Identifies the type of enumeration and the acceptable values in the domain.
Sequencer	Selects one or more instances from a collection using absolute or relative terms such as <i>first</i> , <i>random</i> , <i>every third instance [after x]</i> , <i>seventh through eleventh instances</i> .
Recursor	Includes in the identification of instances those that are owned by the element.
Sorter	Imposes a sequence or priority of instances where none previously existed.

Table 4-2: Business Rule Modifiers

4.1.2 Use Case Action Statement and Business Rule Grammar

We have already discussed how action assertion can invoke actions. 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. We consider two grammars for composition of use case action statements that have been proposed. First, Graham's created his SVDPI task script grammar [Grah95] consisting of:

Subject + Verb + DirectObject + [Preposition + IndirectObject]

Cockburn describes another possible grammar structure [Cock97] consisting of:

[TimeSequence or SequenceFactor] + Actor + Action + Constraints.

Consider the following action statement: Every Friday she sends an overdue notice to each lender who is more than three weeks late.

If we decompose this statement into the component pieces of these two grammars, it would look like this:

```
TimeSequence: every Friday
Actor/Subject: she
Action: sends an overdue notice to each lender
    Verb: sends
    DirectObject: an overdue notice
    Preposition: to
    IndirectObject: each lender
Constraint: who is more than three weeks late.
```

From this example we show how the Cockburn grammar is a superset of the Graham grammar. The Graham grammar decomposes the action phrase the Cockburn defines. The Cockburn grammar, on the other hand, includes two additional components – the time sequence and the constraint. Both of these additional components can also be modeled as business rules, and therefore convey details beyond the basic action. Although they may have usefulness as a narrative use case grammar, they do not belong in the text associated with a use case action in our adaptive use case model. Rather, they reside in the business rule model as facts, conditions, etc.

Facts have multiple ways of being stated. Binary associations of business terms can be expressed in two ways. When many business terms are involved in a fact, the text ordering becomes important. Figure 4-1 shows that the relationship between a set of business terms and facts can be expressed through multiple semantically equivalent text orderings. The business rule model must normalize facts so that they are only represented as a model element once, although they may be presented in several ways.

Within the business rule model, all facts are expressed as text orderings consisting of one or more phrases that express the roles of the business terms. In addition to identifying the sequence of the object role in the text ordering, the phrase also provides the text (with a marker) and specifies the syntactic role of the object role (e.g., subject, object).

4.1.3 Business Rule Patterns

As previously introduced, business rule patterns provide a naming convention to conveniently identify these rule collaborators. Thus, these patterns form business rule templates to which specific business terms can be bound. Table 4-3 presents a business rule pattern language using the same categories as Table 4-1.

Rule Pattern	Problem	Solution
Attribute		
Range	Describing a extent between two points	Define a range extent type with upper and lower bounds and unit of measure
Interval	Quantizing the values of a continuous scale	Define an increment-by value and/or rounding type
Domain	Describing discrete states	Define an enumeration type and values
Type checking	Describing if attribute is compatible with a certain type	Define an enumeration type value and its super-types
Default	Establishing an initial value if null	Define a default value
Collection		
Multiplicity	Describing whether an attribute is mandatory; Describing the upper and lower bounds for the number of collection members; Describing irregular cardinality characteristics.	Define a collection of ranges of type integer (If no lower or upper bound is specified, then the attribute is optional and there are no bounds. If both the lower and upper bound are both 1, then the attribute is mandatory and no intermediate collection mechanism is necessary)
Ordering	Establishing the sequence of elements; Describing a relative property of one member to others	Define a set of actual and/or derived attributes as a sort key and direction; Define a functional restriction, scope (element count), & direction
Unique	Describing permissibility of semantic equivalence for member elements	Define a uniqueness value of type <i>Boolean</i>
Recursive	Describing permissibility and/or meaning of recursive ownership	Define a type and depth restriction for member elements
Meta-attribute	Describing an aggregate property of members	Define an aggregate type (e.g. frequency distribution)
Logical Connections		
Sub-typing	Describing type membership of an entity	Define disjunction between type sets and permissible overlapping of type sets.
Association	Describing an association restriction of an entity	Define constraint between entities using the logical connectives: and, or, if exists, for all
Temporal		
Association	Describing a temporal restriction of one entity to other entities	Define existence constraints through temporal ranges and points
Historical	Describing a temporal restriction of one entity to itself	Define ordering rules (functional restriction, scope, and direction) based on temporal ranges and points.
Meta-attribute	Describing a temporal aggregate property of an entity	Define an temporal aggregate type (e.g. rates, counts)
Transaction		
Unit Of Work	Describing a set of actions that must be performed in their entirety or not at all	Define a collection of actions as an atomic transaction
State Transitions	Describing a restriction on the transition of one state to another for an entity	Define a collective transition type or ordered transitions (e.g. progressive, reinitiating, cycle)
Triggers	Describing actions that must take place when a value from an entity is needed or the state of the entity changes	Define a when needed trigger and/or a when changed trigger
Event management	Describing a mutual dependencies and/or synchronization that is necessary between entities	Define a stimulus-response or event-condition-action structure to capture trigger/prevention action logic
Deferred action	Tracking an action that is prevented from being executed immediately	Define blocking and queuing types
Control		
Priority	Resolving conflicts among competing actions that can occur as a result of an event	Define absolute and relative priority levels to handle conflict resolution

Rule Pattern	Problem	Solution
Timed Execution	Describe actions that must occur at a specific or relative point in time	Define calendar types and extent (duration) subtypes
Routing	Associate a real world entity with a role	Define actor role types and organization structure mappings
Authorization	Describing an authorization for performing an action and defining the action's scope	Define authorization types and actor sub-type mappings
Contingency	Describing an action that must occur during a timed window for execution	Link contingency transaction to window
Enforcement	Describing the strength of a rule enforcement	Define enforcement types (e.g. prevent, notify, log, permit)
Self-Modify	Describing reflective, self-modifying behavior	Map reconfiguration and recompilation of components to events
Derivation		
Algorithm	Describing a mathematical calculation	Define a parameterized algorithm type; define a domain specific language
Production	Describing an inferred fact	Define a formal grammar structure for typing antecedents and consequents for induction and deduction operations
Type Conversion	Describing semantically meaningful type conversions	Define type hierarchy mappings
Type Migration	Managing evolution of type definition	Define attribute mapping between type versions
Interpolation	Deriving a point value from a range given another point value and range	Define an interpolation algorithm type
Protocol		
Contract	Describing the formal syntax for cooperating components	Define ordered sequence of arguments, types and defaults
Alternatives	Describing alternative actions or an extension to a course of actions	Define conditions and variant action sequences
Exceptions	Describing exception actions	Define type for exception handling intervention; Audit failures due to unanticipated conditions
Fire Sequence	Describing sequence of operation or rule invocations	Audit non-deterministic invocations

Table 4-3: Business Rule Pattern Problem/Solution Summary

4.2 Business Rule attributes

As described in Table 4-2, a business rule has three Boolean attributes that capture its state – isTrue, isEnabled, isInvoked. The state of the rule is either unknown, true, or false. If the rule is not enabled, then the other two attributes are irrelevant. When a rule is currently invoked, it is in an atomic action state that will either result in a truth-value of true or false if successful. Otherwise, the rule firing caused an exception.

Unlike the other approaches compared in this paper, 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.

4.3 Business Rule Templates

A business rule template provides a mechanism for parameterized use of business rules. Just as a business rule abstracts a collection of facts, a template abstracts a collection of rules across types or sub-types. Table 4-4 provides an example that combines the business rule patterns and business rule modifiers to create a template that governs a family of business rules. In this example, a

template is defined, then applied against a collection, and finally applied against several sub-types of the collection. This results in four business rules at the meta-level. Each business rule subsequently ties to facts at the operational level (i.e. Russ Hurlbut may not have more than six library items checked out at any time).

Template	A [PatronKind] may not have more than [libraryItemKindCheckedOutMax] [libraryItemKind] checked out at any time.
Collection-Multiplicity Pattern	A patron may not have more than [libraryItemCheckedOutMax] library items checked out at any time. A patron may not have more than six library items checked out at any time.
Discriminator Modifier	A patron may not have more than [libraryItemKindCheckedOutMax] [libraryItemKind] checked out at any time. A patron may not have more than five books checked out at any time. A patron may not have more than four magazines checked out at any time. A patron may not have more than three videos checked out at any time.

Table 4-4: Business Rule Template

4.4 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 to the example in Table 4-4, the business rule template is bound to four rules. Each rule is in the form of an integrity constraint, so it decomposes into a conditional test and an action. Each condition (e.g. ‘a patron has five books checked out’) 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 Logical Connection – 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 actual *UseCaseSegment* model element can be imported by each condition, the only unique construct is each business rule that becomes attached to the condition attribute of a *VariantScenarioAction*.

Either approach will provide a solution at their respective extreme end of behavior variability. A single *UseCaseSegment* utilizing the Logical Connection – 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 what sub-type a patron is by applying one or more business rule conditional tests. These classification rules determine which behavioral rules apply. Consider the following rule:

```
IF patron is tenured faculty,  
THEN 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 Collection-Multiplicity Pattern. Just as *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.

4.5 Binding Business Rules to Structural Models

Facts declare data type associations in the problem space. They must be translated to classes in the solution space through the business rule templates previously discussed. This is most importance to the Attribute and Collection business rule categories. Table 4-5 shows how the rule categories ultimately map to UML elements. The primary function of business rules with respect to their implementation in classes is to establish the minimum requirements imposed on the static structure of application and domain classes in order to meet the needs of all use case scenarios.

Rule Pattern	UML Model Element	UML Diagram
Attribute	DataType	Static Structure Diagram
Collection	Type	Static Structure Diagram
Logical Connections	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

Table 4-5: Business Rule Patterns and UML Model Elements/Diagrams

5. Related Work

Many similar and alternative approaches have already been presented in the previous sections and will not be repeated here. This section is intended to present other research efforts and initiatives that share an affinity with application of use cases in managing domain architecture evolution. Three general areas are presented here: use case model representations, business rules, and communicating agents.

5.1. Use case models representations

The traditional use case representation is an ordered list of step that is structured in some form of template. In [Vemu95], the suggested template headings are: use case, actors, contact person/dept, summary, pre-conditions, description(scenario), post-conditions, exceptions, security exceptions, related use cases, attachments. Use case modeling level approaches are described [ABS95] and [HM95], this use case modeling technique provides different levels of representations targeted to different audiences. In [Harw97], Use case templates are suggested for three contexts: establishing core requirements (conceptualization); develop a model that describes behavior (analysis); create an architecture (design). In [Zorm95], the context and composition of scenarios is described through three relationships: within scenarios, such as objects, measures/types, spatial elements, temporal elements, and behavioral elements; between scenarios, including different viewpoints, scenario evolution, elaboration, disjunction, and composition; and side scenarios that describe a building block's details (e.g. value ranges for a unit of measure).

Use case maps represent an alternative modeling device to express sequences of responsibilities across components and has a causal focus as opposed to a temporal one [Buhr95]. In [Wieg97], a similar approach is taken that describes a 'thread' as is a grouping of modules or classes that implement a specific set of related requirements. Use cases provide a natural mapping of requirements to threads by tracking dependencies among use cases and addresses some aspects of distributed computing.

Another approach deals with elicitation of object oriented state machines through use-cases [ML97]. In [KPW97], the concepts of scopes and episodes are presented to address the connection points between static class model and dynamic use case model. A scenario step is defined as either and interaction within the system, a responsibility to be carried out by some human (primary or secondary actor), or a referenced use case. Each use case is represented by a directed use case graph, each node referring to a scenario step.

A key event dictionary is described as a complementary mechanism to use cases [WF97]. The dictionary consists of a table with each row capturing a key event, detection mechanism, event input data, responses, class data, and access type. Use cases overlay their actions onto the object states until all key event dictionary details have been allocated and the object state machines execute as a well oiled machine in response to external and internal stimuli. Use cases employed to drive the process are easily prioritized by dependence, mission criticality, performance thresholds, customer interest, prototyping capabilities, and phased releases. Details in the key event dictionary are allocated according to the priority level.

5.2. Business Rules

One viewpoint that relates use cases to business rules is that techniques such as requirements gathering use cases are likely to give way to more powerful approaches such as those used in knowledge engineering [FCS97]. Another paper proposes that business rules be documented separately from use cases, which should only focus on actual functional requirements, exception and decision situations [Wieg97]. Industry trend reports suggest that business rules engines will allow separating and codifying business rules during the early phases of systems development. Business users and customers will begin to take ownership of their rules [Seyb96].

5.3. Communicating agents

The relationship between workflow and use cases is still not well defined. The position that we have taken is that they are merely different levels of the same process. This is a slightly different view than that taken in [KMJ97], where a correspondence between IDEF functions (input, control, output, and mechanism/resource), agents and use cases is mapped. Essentially, their view is that a use case groups a behaviorally related sequence of transactions, but the IDEF functions add in control flow. Control output is the only important function with respect to agent oriented systems – corresponding to a (usecase) scenario and (agent's) goals and plans. Based on this line of reasoning, a use case is a workflow without control: $\text{workflow} = \text{use case} + \text{control output}$.

Communicating agents must share a framework of knowledge in order to interpret messages exchanged. Two compatible standards specifications, the Knowledge Query and Manipulation Language (KQML) and Knowledge Interchange Format (KIF), provide a specification for communication. KIF is a computer-oriented language for the interchange of knowledge among disparate programs based on Lisp. It has declarative semantics, meaning an interpreter is not needed for manipulating its expressions. There are four special types of expressions in the language: terms denote objects, sentences express facts, rules express legal steps of inference, and definitions define constants [GF92]. KQML integrates concepts from speech act theory. It is a language for programs to use to communicate attitudes about information, such as querying, stating, believing, requiring, achieving, subscribing, and offering. Each KQML message, referred to as a performative, is intended to perform some action by virtue of being sent [DARP93]. A recent KQML specification proposal relies on a facilitator agent that receives advertisements from other agents announcing the messages that they are committed to accepting and properly processing. Agents can access other agents in the facilitator's domain or other domains either through their facilitator, or directly [LF97]. KQML is also compatible with the CORBA services specification.

6. Conclusions and Future Work

In the previous sections we have established a conceptual foundation for defining a adaptive use case extension to the UML specification. We have shown that formalizing the structure of use case through various model element stereotypes offers several advantages over informal use case

representations. First, varying levels of use case refinement tailored to different stakeholder in the application development process provide appropriate abstractions. Second, adaptive use case parameterization provides a flexible means for enumerating use case scenarios through finer grained control, intuitive linking of interdependent parameters, and simplification of the variation mechanisms. Third, the adaptive frame technology is embedded in the model constructs which facilitates adoption of its use as opposed to learning a new language. The model constructs themselves become a domain specific language for business domain architecture evolution. Finally, through the use of business rule templates, specification of an application can be realized primarily through knowledge elicitation that guides the requirements engineering process. Separating business rules from the use cases through parameterization separates process, structure, and policy so that each can vary independently. This work has consequences for business domain architecture evolution, which should at least consider formalizing use cases and business rule as first class architectural artifacts.

Practical application of this research has shown successful implementation of these concepts is closely tied to the domain engineering maturity level of the organization. An organization that does not have a reuse organization in place will gain little from the formalisms presented here. However, workflow applications in a wide range of business domains can benefit these formalisms provided reuse repositories and configuration management systems exist for linking design and implementation artifacts to the adaptive use cases and business rule templates.

The formalisms presented in this research provide interlocking connections to three other aspects of business domain evolution management. The first aspect is development of a complete set of mapping procedures to tie the various derivations together. This will primarily benefit implementers and testers. The second aspect is extension of these formalisms in order to define a fit assessment model and change costing model to assist project management. The third aspect deals with the knowledge elicitation and requirements engineering process through visual programming tools to assist the capturing of the use case processes and business rule policies.

The specific procedures currently being developed include: 1) converting the adaptive use case to a narrative format that summarizes the important scenarios while suppressing details; 2) converting or promoting a variant scenario to the designated main course of action for the use case; 3) refactoring the action steps of the scenario set; 4) converting back and forth between a use case with multiple actors and workflow dialog consisting of a single actor; and 5) enumerating possible scenarios for a desired level of abstraction that is allowed to vary at each action statement; 6) formal OCL constraints for each model element; and 7) converting the conditions-action pairs in business rule statements into adaptive use case model elements.

The fit assessment and change costing models are being created to tie into the formalisms presented here. The fit assessment model will allow evaluation of specification for a new application developed from the domain architecture. Its primary purpose is to negotiate functionality between the customer and the development team. It is often possible to slightly modify the specifications to realize a significant reduction in overall effort. By highlighting these tradeoffs, this model should assist project management in developing a project plan. When combined with a change-costing model based on reuse metrics, the overall intention will be to generate a more reliable resource, time, and cost estimation.

The visual programming tools being explored are based on the concept of cognitive chunking. Due to the wide range of stakeholders, these formalisms must be suitable to a variety of representation and manipulation mechanisms. There is a need to mix and match text description and iconic depiction techniques as well as controlling the levels of abstraction to limit the amount of

information. A project to investigate alternative presentation and input of adaptive use cases and business rules is being undertaken.

7. References

- [ABS95] Frank Armour & Lorry Boyd & Monica Sood, "Use case modeling concepts for large business system development", OOPSLA workshop -- Requirements Engineering: Use Cases and More, Sunday October 15, 1995, AUSTIN, Texas, <http://www.unantes.univ-nantes.fr/usecase/Contributions/toddHansen.ps>
- [Bass96] Paul Bassett, "Framing Software Reuse: Lessons from the Real World", Prentice Hall, 1996, ISBN 0-13-327859-X)
- [Beed97] Michael A. Beedle "A 'light' distributed OO Workflow Management System for the creation of OO Enterprise System Architectures in BPR environments", OOPSLA-97 conference, <ftp://www.fti-consulting.com/pub/bpr-papers/oopsla.pdf>
- [Buhr95] Ray Buhr, "Use case maps: a new model to bridge the gap between requirements and design", OOPSLA workshop -- Requirements Engineering: Use Cases and More, Sunday October 15, 1995, AUSTIN, Texas, <http://www.unantes.univ-nantes.fr/usecase/Contributions/rBuhr.ps>
- [Chaf96] R. Chafi, "Generic Object Oriented Implementation Design", Ph.D. Dissertation. Illinois Institute of Technology, 1996
- [Cock97] Alistair Cockburn, "Structuring Use cases with goals" – JOOP/ROAD 10(5) Sep '97 and 10 (7) Nov '97, <http://members.aol.com/acockburn/papers/usecases.htm> and <http://members.aol.com/acockburn/papers/uctempla.htm>
- [DARP93] "Specification of the KQML Agent-Communication Language", The DARPA Knowledge Sharing Initiative External Interfaces Working Group, 1993 <http://www.cs.umbc.edu/kqml/kqmlspec.ps>
- [DW97] Desmond D'Souza and Alan Wills, "Component-Based Development Using Catalysis, Draft v 0.8", ICON Computing, <http://www.iconcomp.com>
- [FCS97] Peter Fingar, Jim Clarke, and Jim Stikeleather, "The business of distributed object computing", Object Magazine - 7(2) Apr 1997, pp. 28-33.
- [Fowl96] Martin Fowler, Analysis Patterns : Reusable Object Models, Addison-Wesley, 1996, ISBN: 0201895420
- [Fran97] "Building Knowledge-Based Applications with Dynamic Object Technology", White Paper. Franz, Inc., Berkeley, CA, 1997.

- [GF92] Michael R. Genesereth and Richard E. Fikes, "Knowledge Interchange Format Reference Manual", Version 3.0, Computer Science Department, Stanford University, 1992, <http://logic.stanford.edu/sharing/papers/kif.ps>
- [GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, "Design Patterns: Elements of Reusable Software", Addison-Wesley ISBN 0-201-63361-2, 1994
- [Gott97] Ellen Gottesdiener, "Business Rules show power, promise", Application Development Trends, March 1997, Vol. 4 No. 3. pp36-53.
- [Grah94] Ian Graham, "Migrating to Object Technology", Addison-Wesley, 1994, ISBN 0-201-59389-0
- [Harw97] R. J. Harwood, "Use case formats: Requirements, analysis and design", JOOP 9(8), January 1997, pp. 54-57.
- [HM95] Philip Haynes, Tim Menzies, and Geoffrey Phipps, "Using The Size of Classes and Methods as the Basis for Early Effort Prediction; Empirical Observations, Initial Application; A Practitioners Experience Report", September 22, 1995, OOPLSA '95 Conference Proceedings
- [HH95] David Hay, Keri Anderson Healy, "GUIDE Business Rules Project - Final Report", November 7, 1995 <http://www.guide.org/ap/apbrules.pdf>
- [Holl94] David Hollingsworth, "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>
- [Hurl97d] Russ Hurlbut, "A Survey of Approaches for Describing and Formalizing Use Cases", Technical Report: XPT-TR-97-03, Expertech, Ltd., 1997 <http://www.iit.edu/~rhurlbut/xpt-tr-97-03.html>
- [Hurl97g] Russ Hurlbut, "The Three R's of Use Case Formalisms: Realization, Refinement, and Reification", Technical Report: XPT-TR-97-06, Expertech, Ltd., 1997 <http://www.iit.edu/~rhurlbut/xpt-tr-97-06.pdf>
- [Hurl98a] Russ Hurlbut, "Managing Business Domain Architectures through Use Case Formalisms", Ph.D. Thesis, Department of Computer Science and Applied Mathematics, Illinois Institute of Technology, 1998 (work in process)
- [JCJO92] Ivar Jacobson, Magnus Christenson, Patrik Jonsson, Gunnar Overgaard, "Object-oriented software engineering : a use case driven approach", (New York) : ACM Press ; Wokingham, Eng. ; Reading, Mass. Addison-Wesley Pub.,1992
- [JGJ97] Ivar Jacobson, Martin Griss, Patrik Jonsson, "Software Reuse", ACM Press, New York, 1997 ISBN 0-201-92476-5.

- [KMJ97] Elizabeth A. Kendall, Margaret T Malkoun, and C. Harvey Jiang. "The application of object oriented analysis to agent-based system", JOOP 9(9), February 1997, pp. 56-63.
- [KPW97] Georg Kusters, Bernd-Uwe Pagel, Mario Winter, "Coupling Use Cases and Class Models", Proc. of the BCS-FACS/EROS workshop on "Making Object Oriented Methods More Rigorous", Imperial College, London, June 24th, 1997, pp. 27-30,
<ftp://ftp.fernuni-hagen.de/pub/fachb/inf/pri3/papers/winter/RoomAbstract.ps.gz>
- [LF97] Yannis Labrou and Tim Finin. "A Proposal for a new KQML Specification", TR CS-97-03, February 1997, Computer Science and Electrical Engineering Department, University of Maryland Baltimore County, Baltimore, MD 21250.
<http://www.cs.umbc.edu/~jklabrou/publications/tr9703.ps>
- [Lieb96] Karl Lieberherr, "Chapter 2, Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns", PWS Publishing Company, 1996, ISBN 0-534-94602-X
- [ML97] Ian Mitchell and Hugues Lecoeuche, "On an improved approach to the elicitation of O-O state machines by use-case", JOOP 9(9), February 1997, pp. 52-55.
- [MWFF92] Raul Medina-Mora, Terry Winograd, Rodrigo Flores, Fernando Flores, "The Action Workflow Approach to Workflow Management Technology.", CSCW 92 Proceedings, November 1992.
- [Nass91] E. F. Nassiff, "The Hierarchical Policy Model", MS Thesis, Illinois Institute of Technology, 1991
- [Ross97] Ronald G. Ross, "The Business Rule Book – Classifying, Defining and Modeling Rules, Second Edition" Database Research Group, Boston, MA.
- [Schu94] Stephen G. Schur, Database Factory: Active Database for Enterprise Computing John Wiley & Sons, 1994 ISBN: 0471558443
- [Seyb96] Patricia B. Seybold, "Start your business rules engine", Computerworld 12/09/96
- [Silv95] Mauricio J. V. Silva, "A/OODBMT - An Active Object-Oriented Database Modeling Technique" PhD Thesis, Illinois Institute of Technology, Computer Science Department of Illinois Institute of Technology, 1995
- [UML97] Unified Modeling Language, Version 1.1, <http://www.rational.com/uml>
- [Vemu95] Chandra Vemulapalli, "A Use Case FAQ", OOPSLA workshop -- Requirements Engineering: Use Cases and More, Sunday October 15, 1995, AUSTIN, Texas,
<http://www.unantes.univ-nantes.fr/usecase/Contributions/chandra.ps>

- [VJK96] Vijay Vaishnavi and Stef Joosten, Bill Kuechlerin, "Modeling Workflow Management Systems Using Smart Objects", Proceedings of the NSF Workshop on Workflow and Process Automation in Information Systems, Athens, GA, May, 1996 http://www.cis.gsu.edu/~vvaishna/object_research/projects/smart_object/allsec3.htm
- [WF86] Terry Winograd and Fernando Flores, "Understanding Computers and Cognition: A new Foundation for Design", Ablex Publishing Corporation, Norwood, New Jersey, 1986.
- [WF97] Becky Winant and Mike Frankel, "Solving object state model mysteries using a key event dictionary", JOOP 10(1), March-April, 1997, pp. 52-58.
- [Weis97] Conrad Weisart, "Point-Extension Pattern for Dimensional Numeric Classes", ACM SIGPLAN Notices, Vol. 32, No. 11, November 1997, pp. 17-20.
- [Wieg97] Karl Wieners, "Use Cases: Listening to the Customer's Voice", Software Development, March 1997, Vol. 5., No.3.
- [WFMC96] "Workflow Management Coalition Terminology & Glossary", Document Number WFMC-TC-1011, Issue 2.0, Workflow Management Coalition, Brussels, Belgium, June 1996, <ftp://ftp.aiiai.ed.ac.uk/pub/projects/WfMC/glossary/glossary.pdf>
- [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.aiiai.ed.ac.uk/pub/projects/WfMC/if1/wg1_1000.pdf
- [Zorm95] Lorna A. Zorman, "The context and composition of scenarios", OOPSLA workshop – Requirements Engineering: Use Cases and More, Sunday October 15, 1995, AUSTIN, Texas, <http://www.isi.edu/soar/lorna/oopsla95.ps>

Appendix A – UseCase and Class Meta-Attributes

```
UseCase::Classifier::GeneralizableElement::Namespace::ModelElement::Element
ModelElement.name : Name
GeneralizableElement.isRoot : Boolean
GeneralizableElement.isLeaf : Boolean
GeneralizableElement.isAbstract : Boolean
UseCase.extensionPoint : list of String
```

```
Class::Classifier::GeneralizableElement::Namespace::ModelElement::Element
ModelElement.name : Name
GeneralizableElement.isRoot : Boolean
GeneralizableElement.isLeaf : Boolean
GeneralizableElement.isAbstract : Boolean
Class.isActive : Boolean
```