**RC23365 (W0410-044) October 7, 2004**
**Computer Science**

# IBM Research Report

## Toward Engineered, Useful Use Cases

**Clay Williams, Matthew Kaplan, Tim Klinger, Amit Paradkar**
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

**Research Division**
**Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich**

**LIMITED DISTRIBUTION NOTICE**: This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties). Copies may be requested from IBM T. J. Watson Research Center , P. O. Box 218, Yorktown Heights, NY 10598 USA (email: reports@us.ibm.com). Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home.

# Toward Engineered, Useful Use Cases

Clay Williams, Matthew Kaplan, Tim Klinger, and Amit Paradkar

IBM Thomas J. Watson Research Center
19 Skyline Drive
Hawthorne, NY 10532
{clayw, mmk, tlinger, paradkar}@us.ibm.com

**Abstract.** We explore common problems that exist in the practice of use case modeling: lack of consistency in defining use cases, misalignment between the UML metamodel and the textual representations of use cases expounded in the literature, and the lack of a semantics that allows use cases to be executable and analyzable. We propose an engineering approach to the issues that can provide a precise foundation for use case development. We next discuss four potential uses of such a foundation and identify the research problems that must be addressed to support these applications.

## 1 Introduction

Use cases were developed as a technique for capturing the required behavior of a software system. [6] While they have been successful in having an impact on software development, their usage is not as prevalent as we believe it could be. Several potential factors account for this, including:

1. Confusion regarding the meaning of the term *use case.* Different approaches advocate graphical versus textual methods. Further confusion arises as levels of abstraction vary widely within and across projects using use cases.
2. Misaligned characterizations of use cases in the UML metamodel. In UML, use cases are characterized as BehavioredClassifiers, which mean they have structure and behavior in the form of operations and attributes. This conflicts with best practices for representing use cases that have been proposed. [1,5]
3. Use case semantics are poorly defined, limiting their usefulness to primarily being vehicles for communication. While this is an important use for them, wider industrial use requires that they support machine based processing and analysis.

This paper is a philosophical and conceptual exploration of each of these points in detail in which we identify potential solutions that support an engineering approach to creating use cases. However, engineering use cases is not an end in itself – the technique must provide value across multiple dimensions to be adopted in an industrial setting. To support our approach, we discuss four potential uses of such a foundation (prototyping, estimation, refinement to design, and test generation). We close by discussing related work and future research.

## 2 Use Case Confusion

Across projects, use cases exist at multiple levels of abstraction and are captured in many different notions. The leading practitioners [1,5] all advocate similar content that includes preconditions, main (success) sequences, alternative/exceptional sequences, and postconditions. We believe that this serves as a good basis for an engineered use case representation, but more detail is needed to define what the sequences contain.

Another issue is that it is not uncommon to see use cases used across widely different levels of abstraction, from high level usage of a system to extremely detailed descriptions of system behavior. This lack of consistency regarding the level of use case abstraction is directly influenced by the content supported within a use case model. We take the position that the most useful use cases are those that Cockburn calls "sea level" use cases [5], which are defined at the user-goal level. Our desire is to make constructing good sea level use cases as intuitive and repeatable as possible. A second goal of our representation is to enable the capture of precise, analyzable, and executable use cases with as little modeling and information required as possible.

We start by defining a precise contextual foundation for use cases. Use cases are usually collected during the inception phases of a project. They allow the system requirements to be specified by say *what* the system will do without saying *how* it does it. This requires the capability to talk about the key elements in the system in a very precise way. However, inception occurs prior to architecture definition and design, so we must carefully define the context against which use cases are developed in order to achieve this precision. This context is the domain model of the system.

A domain model is a model of the significant elements from the system's application domain. These are the elements that will be created, modified, and used in the application. They are represented as UML classes with attributes. Also present in the domain model are domain rules (or invariants,) which capture constraints that must hold for instances of the domain objects to be valid. Figure 1 shows the domain model for a simple example ATM application.

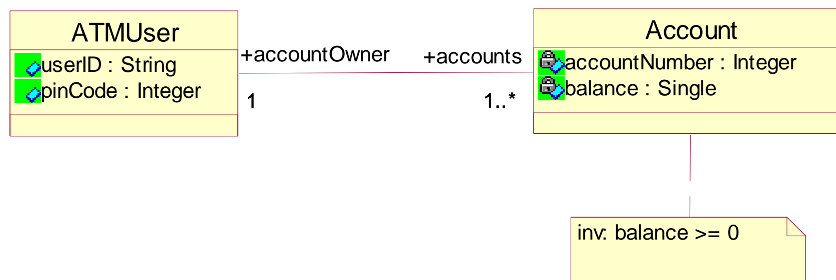


**Fig. 1. The Domain Model for an ATM Application.**

The domain model plus the four primitive types (Boolean, float, integer, and string) serve as the type system for detailing use cases. Use cases are typically defined as sequences of actions that occur between the system and one or more actors, but the specifics of the action types that are supported are left undefined in UML. In

our scheme, use cases support 4 basic actions and 4 flow-of-control actions, each of which has an associated statement for use in detailing the use case.

Basic Actions:

- Input – an actor provides input to the use case
- Output – the use case returns output to an actor
- Computation – a computation is performed using provided input and domain instance information. Computation entails creating, modifying, or deleting instances of domain classes.
- Exception Handling – the system responds to an issue with the input or state of the domain model instances.

Flow of Control Actions:

- Selection – allows the use case to conditionally execute actions
- Iteration – allows the use case to repeatedly execute an action sequence
- Inclusion – allows the use case to include the behavior of another use case
- Extension – defines the extension point for an extending use case

Use cases also have preconditions and postconditions, and both are written in terms of the domain model. Below is an example use case for the Withdraw Money capability for an ATM system.

Use Case: Withdraw Money

Precondition: ATM Customer must be logged onto system.

*1. ATM Customer selects Withdraw.*

2. System requests amount.

*3. ATM Customer enters amount.*

4. System dispenses cash.

**5. System debits amount from the account balance.**

Exceptions:

3.1 [amount greater than account balance]

3.1.1 System displays balance exceeded warning

**3.1.2 If ATM Customer chooses "continue" rejoin at 3**

**3.1.3 Else terminate use case**

Postconditions:

Successful withdrawal:

```
        account.balance = account.balance@pre – amount

    Unsuccessful withdrawal

        account.balance is unchanged.
```

In this example, the input statements appear in *italics*, the output statements are underlined, and the computation statement is in regular **bold** font. The example also illustrates exceptions, selection actions, and rejoining the main scenario from an exceptional alternative.

This representation (use cases, a high level domain model, and domain invariants) serves as a basis for engineered use cases. The use cases consist of very specific types of actions. They specify, in terms of the domain model, what occurs in an interaction with the system being specified without saying how it is done.

## 3 Metamodel Alignment for Engineered Use Cases

### 3.1 UML 2.0 Use Case Shortcomings

Figure 1 shows the current metamodel for UML 2.0 use cases [16].

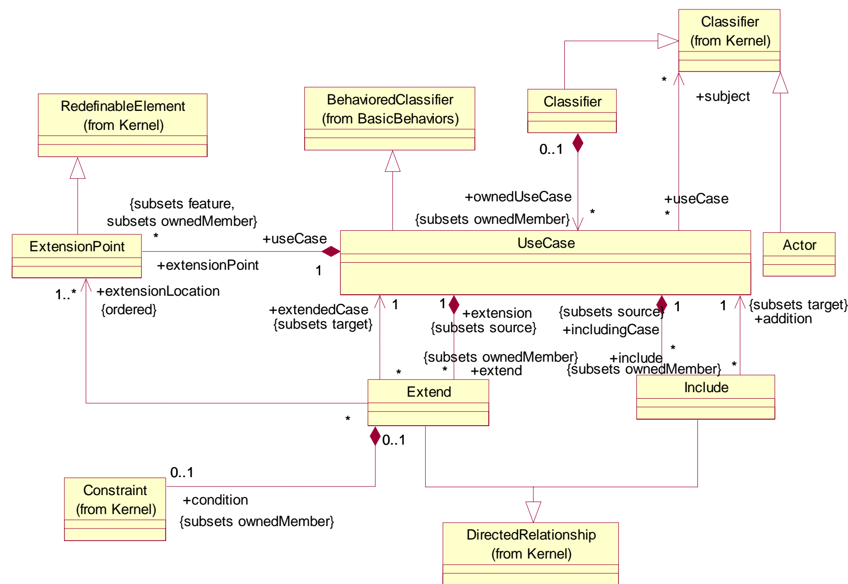


**Fig. 2.** The UML 2.0 Metamodel

Although the UML 2.0 metamodel improved over 1.x by adding the notion of subjects (what we call context in [17]) to use cases, it does contain two main shortcomings.

Given the current size of the UML 2.0 specification, the remedies to these issues are complex, which prevents a complete and formal rectification of them in this paper. In section 3.2, we sketch the approach that we believe needs to be used in order to support remedy them.

#### 3.1.1. UML 2.0 Use Case Misalignment Problems

In the UML 2.0 metamodel, the UseCase metaclass aligns with the BehavioredClassifier metaclass. Classifiers have attributes and features, implying that if they are instantiable, their instances can have state. We believe that as specifications of what a system does rather than how it does it, use case instances are essentially stateless. Of course, to execute a use case requires a program counter to track the current statement/action, but that belongs to the execution environment, not the use case. As we noted earlier, the domain model serves as a type system for the use cases. Instances of the domain classes and their associated state are the basis for describing the effects of the use cases. BehavioredClassifiers also have ownedBehaviors, which misaligns with the typical textual based approaches for use cases.

There a two approaches that could be used to rectify these problems. Use cases could subclass Behavior. In the current metamodel, this is problematic, as Behavior always exists in the context of a BehavioredClassifier and use cases should exist in the context of the domain model, which consists of several classifiers. Another approach would be to add constraints to UseCase to prohibit ownedBehaviors and attributes from existing on use cases. Then classifierBehavior could serve as the basis for specifying the use case behavior. However, this approach has the problem that it presumes that the classifierBehavior is updating the state of the use case instance rather than the state of domain class instances. To properly support alignment between the graphical syntax of UML and the textual details in a use case requires a careful re-examination of the UML metamodel.

#### 3.1.2. Use Case Content is Missing

We argued earlier that use cases should consist of four basic actions (input, output, computation, and exceptions) with supporting actions providing for control flow specification (inclusion, extension, selection, and iteration.) Were it possible to align use cases more carefully with Behavior, refining the model to include specifics regarding contents of a use case would offer a degree of precision that is currently not available. Allowing use cases to support a (possibly enhanced) subset of the existing UML 2.0 actions defined for activities would provide specificity that is lacking in the current specification. As noted above, there is currently no good way to do this in the existing metamodel.

### 3.2 Toward a Metamodel Supporting Engineered Use Cases

As we cannot redesign the use case portion of the UML 2.0 metamodel in this paper without possibly causing problems in other portions of the specification, we instead outline an approach that can be used to address the concerns raised in section 3.1.

#### 3.2.1 Use Case and Use Case Context Realignment

The first area that must be addressed is which UML metaclass the UseCase metaclass aligns with. As suggested earlier, we believe the best remedy is to have UseCase be a specialization of Behavior. This is consistent with the vision defined in section 2, where use cases have input parameters, return results (output), and have preconditions and postconditions. All of these concepts are already supported in the current Behavior metaclass. This requires that Behavior be modified or extended to allow association with multiple classifiers in order to support using the domain model as the use case context.

#### 3.2.2 Use Case Content

Were we to align UseCase with Behavior, the notion of supported actions is still missing. Like the Activities metaclass (which also aligns with Behavior,) the UseCase metaclass should specify a set of Actions which can be performed within a use case instance. A subset of the current actions defined in UML 2.0 would suffice as a basis for creating executable use cases. These include the *Read Write Actions* (Object Actions, Variable Actions, Link Actions, and Structural Feature Actions). These should be extended with specific *Computational Actions* (such as arithmetic, Boolean, and string related actions, and domain specific actions.) *Invocation Actions* are not required, as neither use cases nor their contexts support operations or signals.

Given the alignments proposed in this section, we next explore the semantics required to support engineered use cases.

## 4 Toward Precise Use Case Semantics

In order to support an engineering approach to creating precise use cases, we must address semantic issues in addition to the structural/content issues discussed above. While space considerations prevent a formalized treatment of all relevant semantic issues, we provide informal details and in the following discussion. There are two areas for which we provide specific attention: execution semantics and generalization.

### 4.1 Execution Semantics

For use cases to be precise, we must be able to describe what they mean when they are executed. The execution of a use case is performed in the context of a state consisting of instance of domain model classes. Domain objects may be created, modified, or deleted. Links may be added, or deleted. Attributes may be modified. The following capabilities describe the execution semantics of use cases.

1. Input / Output statements – instantiate formal parameter(s) specified in the statement with value(s) of the appropriate type(s).
2. Computation – create/delete an instance of a domain object, create/delete a link between existing domain object instances, modify the attribute of a domain object.
3. Exception – check exception conditions and fires if true.

Given a use case, execution proceeds as follows. Each statement in the main scenario is executed. During its execution, the alternatives / exceptions defined for that statement are checked for execution eligibility. If one or more is eligible, we must determine which one takes priority and execute it. The semantics for determining priority must be defined in the language. If the alternative specifies where control returns to, that is the point where execution continues, otherwise, it resumes at the next statement in the main interaction course.

While this is only an informal explanation of execution semantics, our group is exploring surface representations for the use case body and will provide a formal semantics based upon that representation in the future.

### 4.2 Generalization Semantics

Generalization has traditionally been a very problematic area for use case semantics. UML 2.0 says nothing about use case generalization, leaving it unclear whether generalization is meant to be a sub-typing mechanism, or a mechanism for simply indicating that the scenarios specified by a parent use case can be replaced with the scenarios specified by its children. If sub-typing is the goal, and use cases are re-aligned to specialize the Behavior metaclass, the design-by-contract calculus can serve as a basis for formalizing generalization. This will ensure the capacity to substitute the child use case for the parent in a context. If scenario replacement is the goal, a careful analysis of the issues related to replacement is required. Of course, a full semantics of use case generalization must be worked out in the context of an updated metamodel that supports both alignment with UML and a precise textual specification for a use case.

## 5 Industrial Applications of Engineered Use Cases

Before engineered use cases will be accepted in an industrial setting, they must provide value that exceeds the effort required to create them. We believe that engineered use cases can provide value across at least four dimensions: prototyping, estimation, refinement to design, and test generation. The first three are in the early research stage in our organization, while the fourth is one in which we have considerable experience.

### 5.1 Prototyping

The first benefit of using precise, engineered use cases is the ability to use them as the foundation for an executable prototype of the system. This allows early feedback to the customer and provides a means for validating the requirements. The prototyping approach requires that a precise surface syntax for specifying the use case content exists, along with an environment capable of executing these use cases. We have developed beta versions of these elements, and have shown that use cases captured with them can be executed directly. It also requires a facility for prototyping UI

elements and associating them with use case. The steps toward building a useful prototype are as follows:

1. Build the domain and use case models.
2. Create mocked-up user interface (UI) elements for the input and output statements.
3. Associate the UI elements with the input and output statements.
4. Execute the prototype with all involved stakeholders, gathering important feedback.
5. Modify the model according to the feedback and repeat the exercise until the requirements are agreed upon by all stakeholders.

The advantage of prototyping based on engineered use cases is obvious – it requires almost no effort beyond normal analysis and design activities to have an executable prototype for validating user requirements.

### 5.2 Estimation

Once the requirements have been validated using the prototyping approach described above, the next step is to determine how much time and cost are involved in realizing the actual system modeled by the use cases. Use cases alone cannot serve as a basis for estimation, but are an important element for determining the functional complexity of the application being developed. Non-functional requirements also play a significant role in determining time and effort.

We believe that our use case approach will serve as a sound basis for estimation, and we are in the very early stages of exploring this idea further. The basic steps in developing such an estimation approach are:

1. Examine existing projects using the engineered use case methods to gather baseline information for an estimation model.
2. Construct the initial estimation model, and apply it to new projects.
3. Gather feedback from the projects, and refine the estimation model.
4. Release the model for general use.

### 5.3 Refinement to Design

Once the estimate has been accepted and the work authorized, the next two activities occur in parallel. The first is to refine the use cases to a detailed design of the system. This is done as follows:

1. The domain model is refined to a full analysis model. This is done by identifying boundary and controller classes that serve as additions to the model classes already present in the domain model. The boundary classes can be inferred from the UI elements discussed in section 5.1.
2. Operations for all of the classes in the analysis model are identified. These are informed by the computation actions in the use case model, as well as required controller and UI computations for the system.
3. The input statements are mapped to operation invocations on the boundary classes.
4. The output statements are mapped to returned parameters from or operation invocations on the boundary classes.

5. Interaction diagrams are used to show how use cases are realized in the system using the mappings defined in steps 3-4 as a basis.
6. Further refinement to a detailed design and implementation can now proceed from the analysis model and use case realizations.

### 5.4 Test Case Creation/Generation

Finally, we must validate that the system constructed from the refinement described in 5.3 actually meets the requirements specified by the use cases. This is done by creating precise test cases from the use case / domain model. Our group has created test generation tools that automatically produce test cases from these models. We have shown a decrease in the effort required to execute the test suite and an improvement in the coverage of the functionality of the system. [17] has specific details and results on our technique. Test cases can also be created by hand. The major steps for testing are:

1. For each use case, ensure that there are test cases for each path through the use case. This can be done using a technique such as the basis paths method [12].
2. For each condition in a use case (either selection or exception), ensure that condition coverage is achieved using a condition coverage technique [3].
3. For each domain rule in the domain model, determine all use cases which update any variable in the domain rule. Determine the interesting ways in which the implementation could be tested to ensure the rule holds, and perform those tests.

## 6 Related Work

Two recent papers explore extensions or alternatives to the standard UML metamodel for use cases. These include [11], which integrates the activity graph metamodel and use case metamodel. In this view, use cases are still stateful classifiers, and the tight link between the actions comprising the use case and the domain model is missing. [13] presents a metamodel that supports refactoring of use case models. This is interesting work, although the metamodel is not aligned with UML, nor does it pay careful attention to the context in which use cases are written.

Another common area for focus is the correct representation of use case structure. [10] focuses on the necessity of being able to correctly and concisely specify alternative courses in a use case. This paper is related to the need to support alternative sequences of statements and exceptions, and defines the differences between these precisely. [9] offers a structured representation for use case content, and a semi-automated approach to translating use cases into sequence diagrams. This contribution is interesting for both representation and refinement purposes.

There is a significant body of work seeking to formalize the content or manner in which use cases are captured. Both [8] and [7] seek to formalize use cases using concepts from Petri Nets. The first paper seeks to represent use cases using the formalism of Constraints-based Modular Petri Nets, while the second uses Colored Petri Nets. While both papers provide an elegant formal basis for use cases and both could be used for use case execution, neither Petri Net approach has been accepted in

industry beyond the embedded/real-time space. Finally, [14] seeks to represent use cases using higher-order logic. The benefits of this approach are an architecture that supports both static and behavioral information. Again, the downside is lack of industry acceptance of formal representations like HOL.

Finally, in the proposed application areas, there is significant work in the testing area [2,4] and the estimation area [15]. The testing work typically uses refinements of use cases to interaction diagrams for generation, which differs from our approach [17]. Estimation work to date has been somewhat disappointing in its accuracy, primarily due to the wide variation in the abstraction levels at which use cases are captured. We hope that our more defined approach will help mitigate these problems.

## 6 Conclusions

We have argued that for use cases to be truly useful for industrial software development an engineering approach to developing them must be used. Such an approach requires first that use cases be well defined, both structurally and behaviorally. To facilitate this definition, we proposed a method where use cases are developed in the context of a domain model and domain rules. The content of the use cases is carefully elaborated using this context.

Next, we proposed an approach for realigning the UML 2.0 metamodel for use cases to reflect the structure and capabilities described above, followed by a discussion of semantics issues associated with use cases. This focused on execution and generalization.

Finally, we presented four application areas that place engineered use cases in the context of the software lifecycle. These are prototyping, estimation, refinement to design, and test creation. We presented a general outline for using engineered use cases in each area.

What we have presented in this paper is a philosophical and conceptual framework for developing engineered use cases. What is lacking is the formalized machinery to ensure that the approach is well grounded and the complete tooling to support the endeavor. Our team is currently exploring these areas, as well as they key application areas defined above. We believe that precise, engineered use cases can be useful across the lifecycle and result in cost effective, reliable software systems.

## References


1. Armour, F. and Miller, G. Advanced Use Case Modeling: Software Systems. Boston: Addison-Wesley, 2001.
2. Basanieri, F., Bertolino, A., Marchetti E. The Cow_Suite Approach to Planning and Deriving Test Suite in UML Projects. In: Proceedings of the 5th International Conferenece on UML (<<UML>>2005), Sept. 30 – Oct. 4, 2002, Dresden, Germany, 383-397.
3. Beizer, B. Software Testing Techniques. Van Nostrand Reinhold, New York, New York, 2nd edition, 1990.

4. Briand, L. and Labiche, Y. A UML-Based Approach to System Testing. In: Proceedings of the 4th International Conference on UML (<<UML>> 2004), October, 2001, Toronto, Canada, 194-208.
5. Cockburn, A. Writing Effective Use Cases. Boston: Addison-Wesley, 2001.
6. Jacobson, I. Object-Oriented Software Engineering: A Use Case Driven Approach. Boston: Addison-Wesley, 1992.
7. Jorgensen, J.B. and Bossen, C. Executable Use Cases: Requirements for a Pervasive Health Care System. In: IEEE Software, vol. 21, no. 2, March-April, 2004, 34-41.
8. Lee, W.J., Cha, S.D, Kwon, Y.R. Integration and Analysis of Use Cases Using Modular Petri Nets in Requirements Engineering. In: IEEE Transactions on Software Engineering, vol. 24, no. 12, December, 1998, 1115-1130.
9. Li, L. Translating Use Cases to Sequence Diagrams. In: Proceedings of the Fifteenth IEEE International Conference on Automated Software Engineering (ASE'00) , September 11 - 15, 2000, Grenoble, France, 293-296.
10. Metz, P., O'Brien, J., Weber, W. Specifying Use Case Interaction: Types of Alternative Courses. In: Journal of Object Technology, vol. 2, no. 2, March-April 2003, 111-131.
11. Nakatana, T., Urai, T., Ohmura, S., Temai, T.: A Requirements Description Metamodel for Use Cases. In: Proceedings of the Eighth Asia-Pacific Software Engineering Conference (APSEC'01), December 04 - 07, 2001, Macao, China, 251-258.
12. Poole, J. A Method to Determine a Basis Set of Paths to Perform Program Testing. National Institute of Standards and Technology, Report #5737.
13. Rui, K. and Butler, G.: Refactoring Use Case Models: The Metamodel. In: Proceedings of the 25th Australasian Computer Society Conference (ACSC 2003), Adelaide, Australia, 2003.
14. Rysavy, O. and Bures, F. Formal Abstract Architecture for Use Case Specifications. In: Proceedings of the 11th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS'04), May 24-27, Brno, Czech Republic, 203-210.
15. Smith, J. The Estimation of Effort and Size Based on Use Cases. IBM/Rational report available at http://www3.software.ibm.com/ibmdl/pub/software/rational/web/whitepapers/2003/finalTP171.pdf
16. UML 2.0 Superstructure Specification. OMG Adopted Specification ptc/03-08-02.
17. Williams, C.E. Toward a test-ready meta-model for use cases. In: Proceedings of the Workshop on Practical UML-based Rigorous Development Methods, Toronto, CA, 2001, 270–287.