Use Cases—Yesterday, Today, and Tomorrow

By Ivar Jacobson

An Introduction

No other software engineering language construct with as much impact as use cases has been adopted so quickly and so widely as use cases have. I believe this is because use cases, although a simple idea, play a role in so many different aspects of software engineering. So many people have asked me how I came up with the use case idea that I will briefly describe it here. I'll also summarize what we have achieved so far with use cases, then suggest a few improvements for the future.

Yesterday: In The Beginning

Use cases have now been around for more than 16 years. When I first used the term in 1986, they were the evolution of work ongoing since 1967.

Getting to Use Cases

It was 1986; I was troubled by how to model telephone calls. A modern switch at that time offered so many types of telephone calls: local calls, outgoing calls, incoming calls, and transit calls. There were many kinds of local calls; and many kinds of outgoing calls: calls to a neighbor switch, to a domestic switch, to an international switch. And, on top of this, each one of these calls could be carried out with different signaling systems.

We had discovered the problem of multiplicity and diversity many years ago. We didn't model each type of call—there were too many, and there was a lot of overlap between them—we just listed and named all of them: we called them traffic cases. *Instead, we modeled the different "functions" we needed to carry out all of the calls.* A function was some loosely defined piece of software. Functions had no interfaces. They had beginnings and endings, but they were not well defined. A function could interact with the outside world. The general feeling was that we didn't really know what functions were, but we could give examples of them, and some people could specify them.

However, we did know how to realize functions. I had learned a diagramming technique that described sequences of relay operations. In 1969 I translated this technique to software to describe component interactions—to what is today called sequence diagrams—the same term used when they were introduced. We described how functions were realized by using sequence diagrams (or collaboration diagrams for simpler interactions) in very much the same way that we describe use case realizations today.

Then, one day in spring of 1986, while working on traffic cases and trying to map them onto functions, I suddenly got it. A traffic case could be described in terms of functions by using an inheritance-like mechanism. I changed the terminology and made *traffic cases* and *functions* both use cases—the former became *concrete* or *real* use cases, the latter became *abstract* use cases.

I wrote a paper on this for OOPSLA'86; this paper is where I introduced use cases. The paper was not accepted (probably because I already had another paper for that conference, or because most people in the program committee were programming language experts). However, the paper was accepted for OOPSLA'87. This paper introduced many of the key ideas in use-case modeling.

What Was a Use Case in 1987?

According to the OOPSLA'87 paper "*a use case is a special sequence of transactions, performed by a user and a system in a dialogue*." This is pretty similar to our current (informal) definition. I developed a separate model for describing an *outside* perspective of a system and I called it a *use-case model*. By outside, I meant a *black-box view* of the system—the internal structure of the system would be of no interest in this model. Some people have misunderstood the term "outside" and believed it to be a synonym for user interface—which it was not. Instead it represented a model of the functional requirements of the system.

At this time the use-case model also included entity (domain) objects, thus we could show how use cases could <<access>> entities. Use cases and entities were class-like (with operations) and not data. The other relation in the use case model was <<built-on>> which was described as "*an extended form of inheritance relation. Multiple inheritances are common."* In fact, the built-on relation was a combination of the generalization relationship and the <<extend>> relation.

Use cases were not just specified, but also designed and tested. "*You create as many processes* [we would today say activities] *as there are use cases. The conceptual model of the use cases is translated seamlessly into a new model showing how each use case is implemented by means of the identified blocks* [a block would today be subsystem, class, or component]." This sounds pretty much like collaborations. "*Each use case is tested separately to safeguard that the system meets the requirements of the user. Please, note that the use cases constitute the key aspect through the entire development activities*."

Sequence diagrams were used to show the interactions among the blocks/components. This is no surprise since sequence diagrams had shown their value in practice for almost twenty years by then.

What Was a Use Case by 1992?

In 1992 the OOSE book, *Object-Oriented Software Engineering—a Use Case Driven Approach,*[1] was published. During 1987 and 1992 the Objectory process had been in practical use by about twenty customers for important new product development. These customers were involved in many different kinds of systems: management information systems, defense systems (pilot, counter measure, C3I), and telecom systems (POTS, mobile). What was presented at OOPSLA´87 was theory, now we had a lot of practical experience behind the idea. Over these five years use cases had matured.

[1] Ivar Jacobson, Magnus Christerson, Patrik Jonsson & Gunnar Overgaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison Wesley. ISBN: 0-201-54435-0. 1992.

Thus, use cases took much of their current shape (syntax and semantics) before 1992. At that time we had use cases, actors, use-case models, the relationships "inheritance" (now replaced by "generalization") and <<extend>>. I didn't like what we today call the <<include>> dependency since I thought it would damage modeling by inviting functional decomposition.

To increase clarity we made it an important issue to distinguish between *a use case* (as a class-like thing), *an instance* of a use case, and *a description* of a use case.

The depth of the use-case model was in its use cases. Each use-case description contained the following:

- a brief description
- a flow of control
- base flows and alternative flows
- subflows (reusable at many places within the same use-case description)
- preconditions and postconditions

However, use cases were more than a requirements technique. Use cases were like the hub of a wheel[2]:

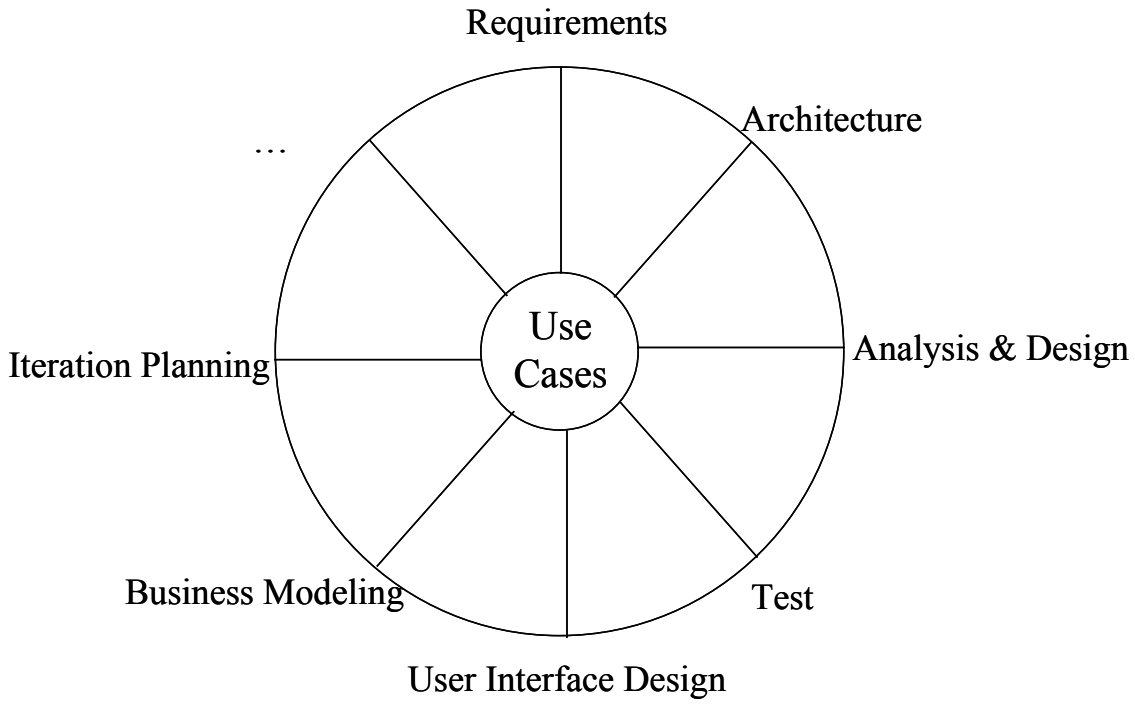

Figure 1. Use Cases Were Like the Hub of a Wheel

[2] In this paper use cases are the focus of our discussion. Their importance in relation to other best practices (architecture first, iterative development) may therefore seem unbalanced.

Use cases were traceable to analysis, to design, and to implementation and test. For each use case in the use-case model we created a collaboration (a view of participating classes) in analysis and design. Each use case resulted in a set of test cases. Use cases were important to design user interfaces and to structure the user manual. Use cases also moved into the space of business modeling, since they perfectly matched the definition of business processes.

We coined the term *use-case driven development* for our approach of software development—first identifying all use cases and specifying each one of them in *requirements*, analyzing and designing each one of them in *analysis and design* respectively, and finally testing each and every one of them in *test*.

We had all this before 1992!

Today: A Lot Has Happened Since Then

The adoption rate of use cases has surprised me: they were embraced almost immediately by all methodologists, and basically adopted worldwide. Other important techniques such as component-based design and object-oriented modeling were much more controversial and needed a much longer adoption time. Probably this is because use cases are basically a simple and obvious idea; they work well with objects and object thinking. Using use cases is not just a technique for managing requirements, but it binds together all the activities within a project—whether this project is a miniproject like a single iteration, or a major project resulting in a new product release.

The current definition of use cases basically goes back to 1994. To strike a balance between defining too many use cases or too few, I added a requirement that **a use case must give a "measurable value" to a "particular actor."** *As a rule of thumb*, I suggested that a large system supporting one business process should have no more than, say, 20 use cases. I realized that giving any such number could lead people to take undesirable actions to get the "right" number. If they had less than 20 use cases, they might split some of them get up to a count of 20; or, if they had more than 20 use cases, they might combine separate use cases to get down to a count of 20. But that is the wrong approach. I have seen good use-case models for commercial systems with as few as 5 use cases and some with as many as 40 use cases. However, I have also seen use-case models with as many as 700 use cases—obviously these were unsound models. **My suggested 20 use cases are concrete (real) use cases and *not* generalizations, or extension/inclusion fragments.**

Use cases have become part of the Unified Modeling Language (UML). Because the UML is precisely defined, the meaning of use cases and associated concepts (such as use-case instance [UCI]) could also be precisely defined thanks to the UML's powerful classifier concept. What I called "class-like" in 1992 could now be formally explained by UML classifiers. We no longer needed to only rely on the old definition that "a use case is a sequence of actions…" Although this is still a compatible definition from the user's perspective, the definition based on classifiers is what methodologists, process engineers, and tool builders need for clarity. *Thus the UML effort resulted in a much more precise definition of use cases, but it didn't do much to evolve them.* Roughly speaking we only changed the "uses" relation to a generalization, and we added

<<include>>. (The "uses" relation in Objectory was previously called "inheritance" and was never intended to be used as <<include>> fragments.) In the past we didn't allow developers to model these fragments, but used another technique involving text objects instead; we'll discuss these later.

I am very happy with the way our Rational Unified Process (RUP) team has correctly implemented use cases and improved their practical use. No really dramatic changes have been made, but use cases have evolved to be better explained, based on the experience of thousands of customers and our own experts. In particular, a new book, *Use Case Modeling*[3], by Kurt Bittner and Ian Spence, is now on the shelves. This is *THE book* on use cases. I strongly recommend everyone involved in software engineering and requirements development read it. Also Kelli Houston's RUP work on user experience design with use cases is a great improvement on our earlier work in this area, and is very much in line with the original use case concept.

Thus, now may be the time to take some steps forward to grow (clarify and extend) the idea of use cases. But first, a word of warning about formalizing use cases.

Use Caution When Formalizing Use Cases

Over the years people have criticized use cases for not having a formal enough description in the UML. Although several of my papers discuss techniques for formalizing use cases, such as using sequence diagrams to show how an actor interacts with a use case, or using activity diagrams or state charts to describe a single use case, I warned against using these techniques. After all*, the use-case model is intended for communicating with customers and users*. Formalizing use cases (using mathematics) has never been a problem. I had already done it in 1986. In fact, any computer science student could do it. By making use cases classifiers in UML, you have the tool to formally describe use cases to basically any depth you want.

The difficulty is, however, to use what is available in UML in the most pragmatic way. *I am still reluctant to suggest that system analysts describe use cases more formally than in some textual form.* Avoid trying to specify the *internals* of each use case with diagrams such as activity diagrams or state charts. You may describe the *interactions* between a use case and actors using sequence diagrams or activity diagrams with swimlanes. I think there are better ways to become more precise about requirements (the internals of a use case) than introducing more formalism in the use-case model. This is the role of analysis—but that is the subject of another paper.

Tomorrow: Potential Next Steps

Over the last decade, the way use cases were written has remained quite stable. Of course, over these years I have wanted to make improvements, however, as soon as a change was discussed, everything was questioned and things became too unsettled. Therefore, I felt it safer to leave it as is, until people become more familiar with the use case construct. Also, we needed to allow time for use cases to be used in the field, and for their implementation to evolve and become established. In this section, I will raise a couple of issues with the current application of use cases, and indicate some proposed changes.

[3] Kurt Bittner, and Ian Spence. *Use Case Modeling.* Addison Wesley Professional, 2003. ISBN 0-201-70913-9

Some Context for the Proposals

A use-case model of a software system contains basically four kinds of use cases:

- **concrete** use cases: can be instantiated (abstract ones can't)
- **generalization** use cases: to support reuse of use cases
- **extension** use cases: add behavior to an existing (or presumed existing) use case, without changing the original use case
- **inclusion** use cases: add behavior to other use cases, and do so by changing them.

Generalizations

These use cases are abstract and **generalizations of concrete use cases (through the generalization relationship) or other abstract use cases**. The generalization use case (the parent use case) and its sub-use cases (children) should be of the same type to obey the principle of substitutability—you should be able to use an instance of a child whenever you expect an instance of the parent. Now, this is not quite true for use cases (or any state-driven classifier), since a child use case can require some extra interaction with the actors. However, the basic idea is the same: the child should be of the same type (classification) as the parent. For example, Make a Local Call and Make a Wake-Up Call are both generalized to the abstract use case Make a Call.

Extensions

Recall that extension use cases serve a very special purpose: they **add behavior to an existing (or presumed existing) use case, and do so without changing it**[4]. Using extensions is a technique to get easy-to-understand descriptions. First you describe the basic (mandatory) behavior, then you add extra (mandatory or optional) behavior—behavior that is not needed to understand more basic behavior. In this way you can start by describing some very simple basic behavior, and then add more and more behavior without having to change the basics. *Extensions are not just a technique for describing optional behavior (optional, that is, from the customer's point of view), they are also intended for describing mandatory behavior in a structured way.* Without a mechanism like extensions the base flow of a use case would become cluttered with statements that have nothing to do with the base use case, even if the statements are important for other use cases.

A potential problem in using extensions is *creating deep hierarchies of extend dependencies*. To avoid these we need guidelines: we *usually never extend an extension (a fragment)*, since it would make them difficult to understand.

[4] Those of you interested in aspect-oriented programming, recognize that the intention of extensions are very similar to the intention of AOP.

Another potential point of confusion is knowing when to use extensions and when to use alternative paths when describing a use case. Again, we need guidelines: we usually only use the extend relationship when the *extension use case is completely separate from the extended base use case*—or, more precisely, when it is a separate concrete use case in itself, or when it is only a small fragment also needed by another use case. *The base use case must be complete by itself and not require the extension. Otherwise, you must use alternative paths to describe additional behavior.*

Extensions can help a lot in managing software development over the entire software development lifecycle. For example, a large class of extensions could be added without requesting regression tests for the base—you would only need to test the extensions and their cooperation with the existing base. Let me qualify this. First, extensions as language constructs need to propagate through design and implementation: they need to be added to the design model, the implementation model, the executable code, and so on. (For further information look up "extends" in the OOSE book.) Second, only extensions that don't access other use cases' objects (more correctly, extensions that don't modify objects shared with other use-case realizations) would, for instance, belong to this class. When such conditions are fulfilled we could prove that some extensions wouldn't be able to damage the existing software.

I proposed the idea of extensions back in 1978 at Ericsson. Developers didn't embrace the idea until 1991. They were first published at OOPSLA 86[5][6]. But the idea had merit: Ericsson even applied for patents to support extensions. Extensions to C++ and to the operating system—in fact, also to the computer architecture—were suggested by our infrastructure team. Extensions would lower the development costs significantly. Further discussion is out of the scope of this paper, but the point is this: *don't think of extensions as useful for use cases only!*

I hope to address in a forthcoming aspect-oriented paper how extensions could propagate through activities other than use-case modeling—activities such as analysis, design, implementation, and test. This was as I said above actually already described in the OOSE book.

Inclusions

When use cases were born, the need for two kinds of reuse was seen. Since I based use cases on object orientation, I saw great value in *subclassing*, and in 1987 introduced what we called the "*inheritance*" relation between use cases. In 1992 we changed this to "*uses*" to make it less "techie" jargon and easier to adopt by analysts. In the UML it was later called "*generalization*." The other reuse need was simply a mechanism for factoring common flows of events (sharing) from use-case descriptions; for cases where we currently use the <<include>> relation. This shared behavior is what I here call an inclusion.

I was very reluctant to introduce a relation like this, since I foresaw people misusing use cases by applying them as they did functional decomposition. In fact, this is one of the threats to use cases today. People misuse use cases by using them to describe *functions* as opposed to *objects,* and then blame the use-case concept for their problem. To get around this, we introduced another

[5] I am happy to send copies to Rational employees.
[6] Ivar Jacobson . "Language Support for Changeable Large Real Time Systems." OOPSLA86, ACM, Special Issue of Sigplan Notices, Vol. 21, No. 11, Nov. 1986.

idea. In the Objectory tool, we supported reuse through "text objects"[7], reusable objects consisting of a piece of text. These text objects could only be changed by the person responsible for the text object, not by someone else using (or reusing) the object.

Text objects could be reused in multiple places, for example in different text descriptions of use cases. Like Rose/XDE, which has a model element for each class that could be shown (differently if needed) in multiple diagrams, text objects could be shown in multiple documents. The beauty was that all the text objects were kept in one place, so they were easy to find, change and manage, and all references were automatically kept in sync. Very powerful!

I think that the solution we had was right at that time, when we feared that use cases would be viewed as just another way to do functions. This problem persists today among system analysts, although not among methodologists.

Two Classes of Extension/Inclusion Use Cases

When extension use cases originally were introduced I had basically only one kind of extension or inclusion in mind: the *small reusable use-case fragment*. I didn't foresee the need for being able to extend or include concrete complete use cases. This is something we learned during the first four years of practical use. Many times we[8] discussed the need for two kinds of extension use cases. However, we didn't want to make use-case modeling more complex. During the UML 1.1 work we (primarily Jim, Gunnar and I) touched on this subject, but for the same reason we didn't follow through. Maybe now is the time?

There are two classes of extension/inclusion use cases:

1. ones that are *concrete (complete) use cases* in themselves
2. ones that are just *fragments of a use case*

Extensions/inclusions that are concrete use cases in themselves. These use cases interact with actors and can be said to provide value to an actor. As an example[9], consider a "surveillance" system that reports intruders. The base (concrete use case) monitors the surveillance area, and perhaps even does some other work, such as maintaining a constant building temperature. The extending use case (also a concrete use case) reports unusual events—security breaches or fires—to the appropriate authorities (police, fire, building management). This illustrates that the base use case has some significant behavior, as does the extending use case—they are both concrete use cases, and they can both be instantiated.

Extensions/inclusions that are just fragments of a use case. This is a far larger class of use cases, but each member is usually very small. These use cases are abstract in that they *cannot be instantiated separately*. They are needed by some other use case—usually a concrete or a generalization use case. For example, in Figure 2, assume that the base use case is a bank transaction *Conduct Transaction*. Every time a transaction fails the bank wants to register this event to make it available to some *other concrete use case,* in this case, an administrative use

[7] We called them text items.
[8] At this time , we were primarily Gunnar Overgaard, Patrik Jonsson, Karin Palmkvist, and myself.
[9] The example is provided by Kurt Bittner. Thanks.

case *Inspect Transaction Failures*. One obvious traditional solution would be to change the transaction use case and thus explicitly in its description show that a failure message has been registered. However, this would require changing the base and complicating its understanding. The change has nothing to do with the base use case *Conduct Transaction*; it's only there to register some information to the other use case. One such change may not be disturbing, but when you have several of them it gets to be quite messy. To avoid cluttering the base we instead use an extension use cases to add the change *on top of* the base use case. We would add *a third use case*—a very small use case fragment called *Register Failures*. In a similar way we may have small procedure-call like inclusion use cases that are shared between two or more concrete use cases. Assume that *Validate User* is such an included use case fragment (shared with some other real use case).

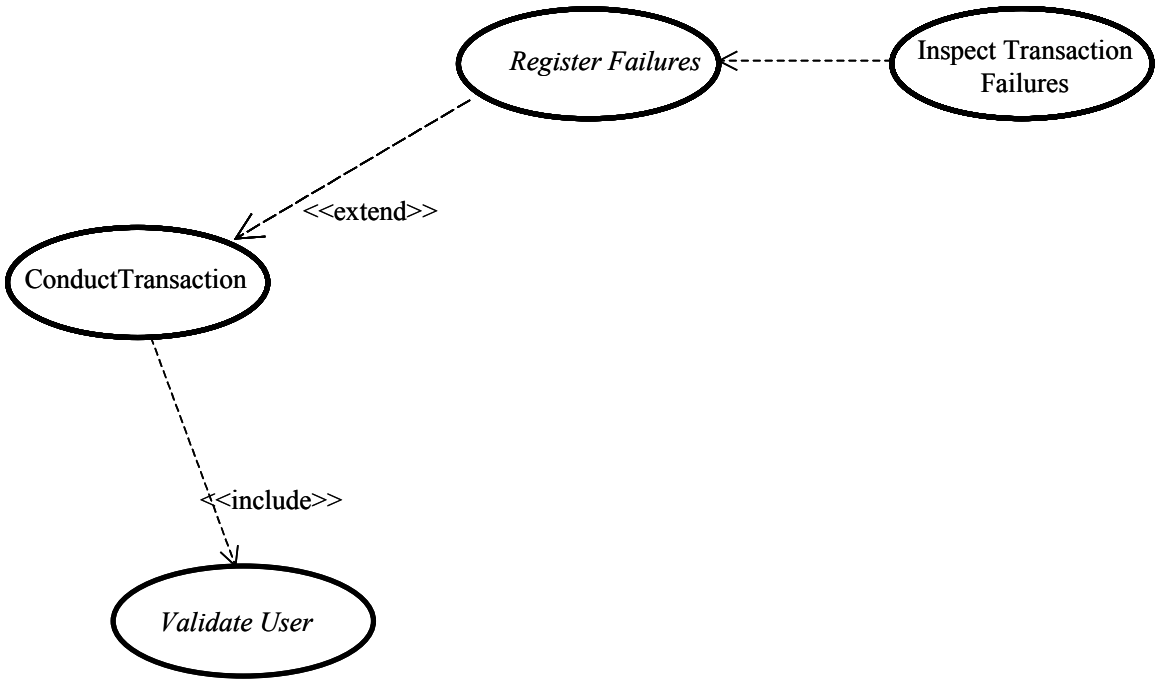

Figure 2. Concrete and Abstract Use Cases

A part of a use-case model with two concrete use cases: *Conduct Transaction* and *Inspect Transaction Failures;* and two abstract use cases (fragments): *Register Failures* and *Validate User.*

I have introduced a dependency between the concrete use case *Inspect Transaction Failures* and the extension use case *Register Failures*. Since this dependency is special and required only to attach an extension fragment to the use case that needs it, it may be a good idea to define a unique dependency stereotype (maybe <<need>>?) for it. But that is a separate issue.

The first class of use cases—*concrete use cases in themselves*—is fine, we don't need to do anything special about it. Concrete use cases can extend a base use case or be included in a base use case.

The second class of use cases—the *use case fragments,* as I will call them—will be discussed below and some changes will be suggested. However, before doing so let's discuss a related subject: the relation between extension and inclusion use cases.

Extension and Inclusion Use Cases Have a Lot in Common

Extension and inclusion use cases are *both related to a base use case*. During the "execution" of the base use case, that is when a use case instance of the base runs, it will (under certain conditions) interrupt the flow described in the base use case and instead follow the flow as specified by the extension or the inclusion use case. When the use case instance has come to the end of the extension or the inclusion use case, it will return to the base use case and to the position in the flow described in the base use case, where it left off. This is true for both complete use cases and fragment use cases.

The major difference between extension and inclusion use cases is the way the use case instance is instructed to interrupt the base use case and instead follow the extension or inclusion use case.

- *In the case of **inclusion**, the base flow itself explicitly instructs the use case instance to obey the inclusion use case.*

- *In the case of **extension**, the base flow doesn't specify the interruption, rather the extension use case specifies where in the base use case the use case instance shall make the interruption*.

  The extension use case references an **extension point**, which specifies a unique location in the base use case. In OOSE and Objectory, extension points belonged to the extending use case. In the work on UML 1.1 Jim Rumbaugh suggested that extension points should belong to the extended use case. The argument was for encapsulation—the extending use case should not see the details of the base use case, just the extension points. If you change the extended use case, only that use case would know the new location. I agreed with him.

Thus extension and inclusion use cases are very similar; in fact, they could be considered *the inverse of each other*. Actually, when working on UML 1.1, Jim and I discussed that this should be reflected in how the two dependencies were named. However, he was not crazy about my proposal that <<include>> should be named <<inverse extend>>, and I don't think anyone else would have been either. ☺

Fragments are Not Use Cases

Neither extension use-case fragments nor inclusion use-case fragments are use cases, and they should not be treated as use cases. Cleaning up this "defect" has long been overdue. It means that we need to make some minor changes to UML, by adding some minor notational elements.

Methodology users have the right to question any new notation. In my own experience I have struggled with "homegrown" methodologies that introduce notation for everything. Most of these methodologies didn't differentiate between syntax (notation) and semantics, and their authors had no education in classic language design—programming language or modeling language.

Thanks to the work on SDL in 1981, and now UML, we have come a long way in developing more precise modeling languages.

Very simply: Classical language specifications (1) start from a *concrete syntactic construct* (a notational element in UML), which is (2) mapped into an *abstract syntactic construct*, and which in turn is (3) mapped onto a *semantic element*. The semantics specifies the meaning of the syntax. Most interesting syntactic constructs have a unique semantic correspondence. (The opposite is not necessarily true, since designed languages usually have many semantic elements [dynamic semantics] that don't have a syntactic correspondence.) Thus I think it is standard language design practice to make every unique semantic element mappable from a unique syntactic construct. Natural languages are much more complex, but since we are creating the UML language ourselves, we don't need to complicate things.

Since fragments are NOT use cases, they should NOT be represented by the use-case syntax. It makes it harder for analysts to distinguish between important elements. Fragments should be treated as they deserve to be treated—as less important than real use cases.

Note, I have no good proposal for what the new notational elements should look like. To suggest something, I have chosen an icon that indicates a tiny element—a dot. However, an icon that indicates a fragment would be more intuitive. Figure 3 is an example of what I mean.

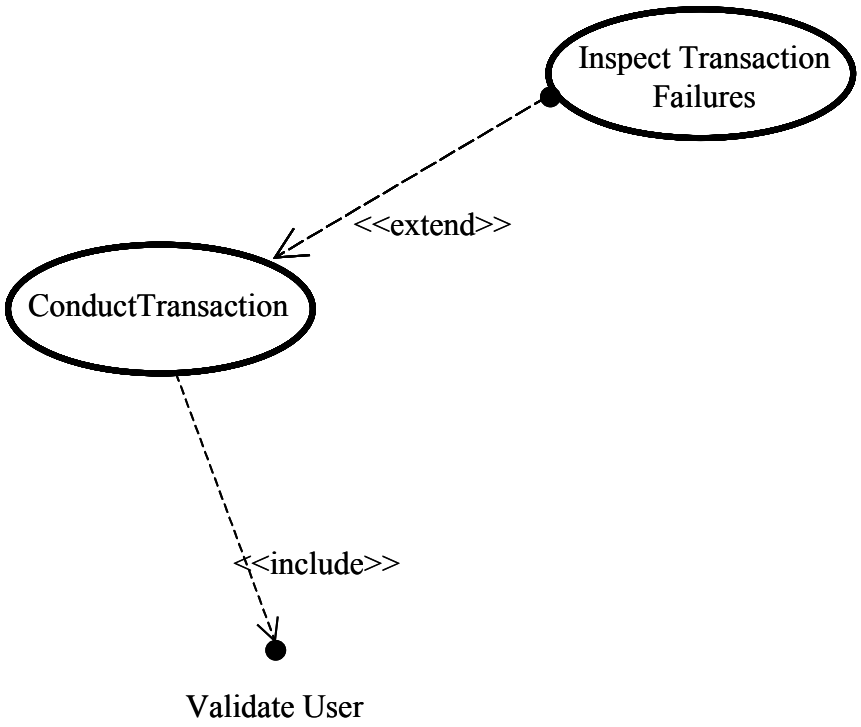

Figure 3. Treat Fragments Appropriately—As Tiny Elements

The *Register Failures* fragment (see Figure 2) has collapsed to a dot and its name has disappeared. The semantics of the dot would be that while executing *Conduct Transaction*, and when reaching the extension point, something specified by the dot will happen: an extension will be executed. The *Inspect Transaction Failures* use case will use the extension to perform its responsibilities.

The dot represents an ***extension fragment***—the *Register Failure* fragment—let's call it "E." E is not part of the real use case *Inspect Transaction Failures* (called UC2) that needs the extension. Since E is only needed by UC2 we don't need to give it a name. Instead we attach it to UC2 by attaching the dot to the use case symbol. However, it must be clear that the use case (UC2) that needs the extension E doesn't <<extend>> the other base use case *Conduct Transaction* (UC1) or the extension E. Thus it would be completely wrong from a language point of view to have use case UC2 <<extend>> UC1. Without the dot or something similar we wouldn't be able to properly explain the desired semantics.

The dot would be expanded (by "clicking the dot") to a new compartment of the use case. We could name the new compartment "Extensions," and use it to describe the extensions needed by the use case.

In the example we only have one extension (*Register Failures*) needed by *Inspect Transaction Failures*. Thus in the extension compartment of *Inspect Transaction Failures* we would describe the *Register Failures* use case. Since we only have one extension, we didn't name it, but if there were multiple extensions, we might name the dots.

Sometimes an extension fragment is needed by several use cases (of type UC2). In these cases the extension fragment must be named uniquely within the use-case model. The dot representing the extension fragment must be "free" from one particular use case, but related (via dependencies—preferably using the new stereotype <<need>> or something similar) to all the use cases that need it. An extension fragment of this kind is a kind of classifier with an extension compartment.

Figure 3 also has an ***inclusion fragment***—the *Validate User* fragment. Inclusion fragments are not real use cases; they are reusable pieces of use-case descriptions or "text objects." Inclusion fragments are elements separate from the use cases that include them, thus they are semantically similar to extension fragments needed by several real use cases.

There is an important difference between inclusion fragments and extension fragments:

- *An* inclusion fragment *will be "executed" by the use case instance that also "executes" the real use case (the one that includes it).*

- *An* extension fragment *will be "executed" by a use case other than the use case instance that needs it.*

With UML terms, a fragment should be a classifier with a notation that makes us think about tiny things—but now I go too deep for the purpose of this paper. We also need a syntactic shortcut (syntactic sugar) to attach a fragment to a concrete use case. We still need to use fragments in a pragmatic way.

The Day After Tomorrow: The Future of Use Cases

Over the years there have been many ideas for improving use cases. There are so many ideas and I don't know of them all, but I'll list some of them below, and introduce some others. I can't refrain from discussing the ones I am most excited about: making use cases code modules through aspect-oriented programming and making use cases run-time entities. Here are possible directions for the "future" of use cases:

- **Stereotype use cases**
  There are many ways to classify use cases. There are primary, secondary, etc. use cases; there are business use cases, software use cases, system use cases, etc.; business use cases are supporting, managerial or operational[10], … Stereotyping is a UML mechanism for classification which would help developers in modeling of use cases.

- **Clarify relationship between patterns and use cases**
  Many design patterns are "template" realizations of reusable use cases. Such patterns are usually described using sequence diagrams or collaboration diagrams. A pattern is a solution to a general problem and it can be applied in different contexts. There is thus an interesting relationship between a pattern and the generic, reusable use case that specifies the problem. Clarifying this relationship would be very helpful to developers.

- **Using use cases within Human Computer Interaction (HCI)**
  HCI is a science. There is a way of designing a user experience by understanding the user community, its habits, and its metaphors. I was introduced to this technology by working with companies that were developing large commercial Web sites for huge user communities. Use cases would be an important concept to integrate software development and HCI approaches.

- **Cost estimations based on use cases**
  In 1994 Magnus Christerson lead Gustaf Karner's master's thesis[11], which resulted in a paper on project estimations based on use-case points (derived from function points). To my knowledge this is still an interesting paper. With all the experience we have today about use cases and project estimations, we should be able to modernize these ideas.

[10] In the *Object Advantage* book we described them in layers. *The Object Advantage: Business Process Reengineering with Object Technology, by* Ivar Jacobson, Maria Ericsson, and Agneta Jacobson. Addison Wesley Longman, 1994.
[11] Gustav Karner, "Use Case Points - Resource Estimation for Objectory Projects." Objective Systems SF AB, (copyright owned by Rational software) 1993.

- **Reusable use cases**
  This is a huge topic. Reuse of business software should start from understanding its use cases—both the use cases of the business, and the use cases of the software to be used. This is the depth of the *Software Reuse* book[12] that I wrote with Martin Griss and Patrik Jonsson back in 1994-97. It is more relevant than ever today when a company's IT support is built by integrating enterprise applications, whether these are legacy systems, packaged solutions, new applications, or Web services. Further discussion can be found in my RUC 2002 talk "Closing The Gap: Business Driven Enterprise Application Integration." (I would be happy to send this to Rational employees.)

Making Use-Case Scenarios First-Class Citizens

It would be helpful to be able to identify and enumerate use-case scenarios, and to show dependencies between these scenarios. Recall that *a scenario is a use-case instance that we choose to model*. This is probably more of a process issue than a language issue. There are probably many kinds of dependencies.

Iteration Dependencies

Each project iteration is driven by a number of use-case scenarios. Usually a use case is not completed within a single iteration, but is worked on over several iterations. I would like to be able to show how iterations are made up of scenarios, how a scenario grows over several iterations, and how several different scenarios over several iterations together make up a complete use case. You should be able to show, for instance, that you may have to develop less important scenarios first just to be able to develop more important instances later. As a concrete example, you may need to first develop a use-case scenario that allows a telephone system operator to make a subscriber a valid user, before that subscriber can make any telephone calls.

Test Case Dependencies

There are other reasons for dependencies between scenarios. One is for testing. Integration testing is built up test case by test case in very similar way to how iterations are built up. We would be able to trace use-case scenarios to test cases. A good use-case scenario is a good test case. The relationship between a use-case approach and a test-first approach would become more streamlined.

Use Cases and Aspect-Oriented Programming

One of the most exciting "new" movements today is Aspect-Oriented Programming (AOP). AOP was the buzzword of the year at OOPSLA. And for very good reasons. This article cannot get into any depth about AOP. However, I can't refrain from giving some hints why AOP will make the future of use cases fantastic. The whole idea with extension use cases, that is to **add behavior to an existing system without changing it,** is very similar to the whole idea of aspects. Use-case realizations are implemented as aspects. Extension use cases will be realized as aspects. Extension points are semantically similar to join points.

[12] Ivar Jacobson, Martin Griss, & Patrik Jonsson. *Software Reuse: Architecture, Process and Organization for Business Success.* Addison Wesley Longman, 1997.

When using the RUP, within each iteration we specify use cases, we design them, and we test them. Between design and test we must disrupt the use-case flow in order to design, code and test the components that together realize the use cases. Using AOP will simplify this: we'll go directly from use-case design to use-case programming and then to use-case test. The work on components will be supported by our programming environment (an AOPL = an OOPL + aspects). Thus AOP allows us to seamlessly implement use cases.

Neither use-case driven development, nor aspect-oriented programming are silver bullets. They represent two best practices only. However, I believe that integrating them will dramatically improve the way software will be developed.

Making Use Cases Run-Time Entities

The most exciting future of use cases is that they will also have counterparts in the run-time environment. Being able to identify executing use cases (use-case instances, in fact) in the operating system can help us with many important features as discussed in my 1985 thesis[13], which had a semantic construct representing a use-case instance. A use-case instance was created by an external event, it lived during the whole interaction with the user (e.g., during the telephone call), and it tracked all the objects that participated in the use-case instance. With such a construct, we could change installed software much more incrementally—one use case instance at a time. The software system could be restarted in much smaller steps, in most cases by restarting only the use-case instance that was broken. And, we could simplify the programming of use cases. With AOP we may achieve parts of this. It seems as we will at the least achieve use-case oriented programming. ☺

Final Words

Use cases have now been around for more than 15 years. We can move them a step forward by cleaning up some minor defects: by separating use cases and fragments. This can be done tomorrow.

The day after tomorrow, there are many interesting ideas to expand on use cases. Many of the ideas we have are just marginal improvements. However, two of the ideas are dramatic enhancements: making use cases code modules and making them executable run-time entities. Until then, enjoy use cases as they are today!

Acknowledgements

I would like to thank Kurt Bittner, Gunnar Overgaard, Paul Szymkowiak for their feedback on an early version of the paper. Thanks to Catherine Southwood for editing the paper.

[13] Ivar Jacobson. "Concepts for Modeling Large Real Time Systems." Dissertation, Department of Computer Systems, The Royal Institute of Technology, Stockholm, September, 1985.