Appendix B

Changes between UML Versions

When this book first appeared on the shelves, the UML was in version 1.0. Much of it appeared to have stabalized and it was in the process of OMG recognition. Since then there have been a number of revisions. In this appendix I describe the significant changes that occur, and how they affect the material in this book. If you have an earlier printing of the book, this summarizes the changes so you can keep up to date. I have made changes to the book to keep up with the UML, so if you have a later printing it describes the situation as it was at that time.

Revisions in the UML

The earliest public release what came to be the UML was version 0.8 of the unified method. It was released for OOPSLA (October) 1995. It was called the "Unified Method" and was the work of Booch and Rumbaugh, as Jacobson did not join Rational until then. In 1996 they released a 0.9 and a 0.91 version that included Jacobson's work. At this time they changed the name to the UML.

Version 1.0 of the UML was submitted to the OMG Analysis and Design Task force in Janurary 1997. It was then combined with other submissions and a single proposal for the OMG standard was made in

Septemember 1997, this was called version 1.1. This was adopted by the OMG towards the end of 1997. In a fit of darkest obfustication the OMG called this standard version 1.0. So it was both OMG version 1.0 and Rational version 1.1, not to be confused with Rational 1.0. In practice everyone calls that standard version 1.1.

UML 1.1 had a number of minor visible changes from version 1.0.

When the OMG adopted UML 1.1 they set up a Revision Task Force (RTF) chaired by Cris Kobryn to tidy up various loose ends with the UML. They interally released version 1.2 in July 1998. This release was internal in that 1.1 remained the official UML standard. You could think of version 1.2 as a beta release. In practice this distinction hardly mattered as the only changes in the standard were editorial: fixing typos, grammatical errors and the like.

A more significant change occurred with version 1.3, most notably affecting Use Cases and Activity Diagrams. The amigos' user guide and reference manual were published late in 1998 with the 1.3 changes, before the official 1.3 documents were made public, which caused some confusion.

In April 1999 the RTF will submit version 1.3 to the OMG as new official standard of the UML. The OMG Analysis and Design Task Force will then take over the UML again and consider any future moves. Of course this is what I currently know, check my web site for future update information.

Changes in UML Distilled

As these revisions go on, I've been trying to keep up by revising UML Distilled with subsequent printings. I've also taken the opportunity to fix errors and make clarifications.

The 1st through 5th printings are based on UML 1.0. Any changes between these printings were minor. The 6th printing took UML 1.1 into account, (however due to a glitch the inside covers still show 1.0 notation). The 7th through 10th printings were based on UML 1.2. Those printings based on UML after 1.0 have the UML version number on the front cover. (Unfortunately a printing error meant that some copies of the 10th printing were labelled as 1.3 — I'm sorry about that.)

In the rest of this appendix I'll summarize the two major changes in the UML, from 1.0 to 1.1 and from 1.2 to 1.3. I won't discuss all the changes that occur, but rather only those that

- change something I said in *UML Distilled*, or
- represent important features that I would have discussed in *UML Distilled*

I am continuing to follow the spirit of *UML Distilled*: to discuss the key elements of UML as they affect the application of the UML within real world projects. As ever, the selections and advice are my own. If there is any conflict between what I say and the official UML documents, the UML documents are the ones to follow. (But do let me know, so I can make corrections.)

I have also taken the opportunity to indicate any important errors or omissions in the earlier printings. Thanks to the readers who have pointed these out to me.

Changes from UML 1.0 to 1.1

Type and Implementation Class

On page 55 of *UML Distilled*, I talked about perspectives, and how they altered the way people draw and interpret models, in particular class diagrams. UML now takes this into account by saying that all classes on a class diagram can be specialized as either types or implementation classes.

An **implementation class** corresponds to a class in the software environment in which you are developing. A **type** is rather more nebulous; it represents a less implementation-bound abstraction. This could be a CORBA type, a specification perspective of a class, or a conceptual perspective. If necessary, you can add stereotypes to differentiate further.

You can state that for a particular diagram, all classes follow a particular stereotype. This is what you would do when drawing a diagram from a particular perspective. The implementation perspective would

use implementation classes, while the specification and conceptual perspective would use types.

You use the realization relationship to indicate that an implementation class implements one or more types.

There is a distinction between type and interface. An interface is intended to directly correspond to a Java or COM style interface. Interfaces thus have only operations and no attributes.

You may use only single, static classification with implementation classes, but you can use multiple and dynamic classification with types. (I assume this is because the major OO languages follow single, static classification. If one fine day you use a language that supports multiple or dynamic classification, that restriction really should not apply.)

Complete and Incomplete Discriminator Constraints

On page 78 of previous printings of *UML Distilled*, I said that the {complete} constraint on a generalization indicated that all instances of the supertype must also be an instance of a subtype within that partition. UML 1.1 defines instead that {complete} indicates that all subtypes within that partition have been specified, which is not quite the same thing. I have found a lot of inconsistency on the interpretation of this constraint, so you should be wary of it. If you do want to indicate that all instances of the supertype should be an instance of one of the subtypes, then I suggest using another constraint to avoid confusion. Currently, I am using {mandatory}.

Composition

In UML 1.0, using composition implied that the link was immutable (or frozen; see below), at least for single-valued components. That constraint is no longer part of the definition.

Immutability and Frozen

UML defines the constraint {frozen} to define immutability on association roles. As it's currently defined, it doesn't seem to apply it to attributes or classes. In my practice, I now use the term frozen instead

of immutability, and I'm happy to apply the constraint to association roles, classes, and attributes.

Returns on Sequence Diagrams

In UML 1.0, a return on a sequence diagram was distinguished by using a stick arrowhead instead of a solid arrowhead (see page 104). This was something of a pain, since the distinction was too subtle and easy to miss. UML 1.1 uses a dashed arrow for a return, which pleases me, as it makes returns much more obvious. (Since I used dashed returns in *Analysis Patterns*, it also makes me look influential.) You can name what is returned for later use by using the form "enoughStock := check()".

Use of the Term "Role"

In UML 1.0, the term **role** primarily indicated a direction on an association (see page 57). UML 1.1 refers to this usage as an **association end**. There is also a **collaboration role**, which is a role that an instance of a class plays in a collaboration. Many people still use the term role to mean a direction of an association, although association end is the formal term.

Changes from UML 1.2 (and 1.1) to 1.3

Use Cases

The changes to use cases involve new relationships between use cases. 1.1 has two use case relationships: «uses» and «extends», both of which are stereotypes of generalization. 1.3 offers three relationships.

- «include» - a stereotype of dependency. This indicates that the path of one use case is included in another. Typically this occurs when a few use cases share common steps. The included use case can factor out the common behavior. An example from an ATM machine might be that "Withdraw Money" and "Make Transfer" both use "Validate Customer". This replaces the common use of «uses».

- generalization (no stereotype). This indicates that one use case is a variation on another. Thus we might have one use case for Withdraw Money (the base use case) and a separate use case to handle the case where the withdrawal is refused due to lack of funds. The refusal could be handled as a use case that specializes the withdrawal use case. (You could also handle it as just another scenario within the Withdraw Money use case.) A specializing use case like this may change any aspect of the base use case.
- «extend» - a stereotype of dependency. This provides a more controlled form of extension than the generalization relationship. Here the base use case declares a number of extension points. The extending use case can only alter behavior at those extension points. So if you are buying a product on line, you might have a use case for buying a product with extension points for capturing the shipping information and capturing payment information. That use case could then be extended for a regular customer where this information would be obtained a different way.

One of the confusions of all this is the relationship between the old relationships and the new. Most people used «uses» the way the 1.3 «includes» is used, so for most people we can say that «includes» replaces «uses». People used 1.1 «extends» in both the controlled manner of the 1.3 «extends» and as a general overriding in the style of the 1.3 generalization. So you can think that 1.1 «extends» is split into the 1.3 «extend» and generalization. Now although this explanation covers most usage of the UML that I saw, it isn't the strictly correct way of using those old relationships. However most people didn't follow the strict usage and I don't really want to get into all that here.

Activity Diagrams

At 1.2 there were quite a few open questions on the semantics of Activity Diagrams. So the 1.3 effort did quite a lot of tightening up on these semantics.

For conditional behavior the diamod shaped decision activity can now be used for a merge of behvior as well as as a branch. Although neither branches or merges are necessary to describe conditional behavior, it is increasingly common style to show them so that you can bracket conditional behavior.

The synchronization bar is now referred to as a fork (when splitting control) or as a join (when synchronizing control together). You can no longer add arbitrary conditions to joins. You must also follow matching rules to ensure forks and joins match up. Essentially this means that each form must have a corresponding join that joins the threads started by that fork. However you can nest fork and joins, and you can eliminate forks and joins on the diagram when threads go directly form one fork to another fork (or one join to another join).

Joins are only fired when all incoming threads complete. However you can have a condition on a thread coming out of a fork. If that condition is false, then that thread is considered complete for joining purposes.

Multiple Triggers are no longer present. Instead you can have dynamic concurrency in an activity (shown with a * inside an activity box). Such an activity may be invoked several times in parallel. All its invocations must complete before any outgoing transition can be taken. This is loosely equivalent, although less flexible, to a multiple trigger and matching synchronization condition.

These rules reduce some of flexibility of activity diagrams, but they do ensure that acitivity diagrams are truly special cases of state machines. The relationship between activity diagrams and state machines was a matter of some debate in the RTF. Future versions of the UML (after 1.3) are expected to make activity diagrams a completely different form of diagram.