
An Introduction to the UML and the Unified Process

3.1 Introduction

This chapter introduces the Unified Modeling Language (UML) notation, its motivation and history. It then presents the Unified Process as a design method, supported by the UML notation, for designing object-oriented systems.

3.2 Unified Modeling Language

The Unified Modeling Language (or UML) was an attempt to bring together the best of the notations currently in use in the early 1990s. It was developed by Rational Corp., originally by Grady Booch and James Rumbaugh. In the early days of UML (and in particular when I first came across it) it was part of the Unified Method, and indeed the first document I read about the UML was actually entitled “Unified Method 0.8”. The intention was to produce not just a notation but a best practice method as well. However, producing a notation is one thing; producing a design method is quite another. Therefore the Unified Method developed into the Unified Modeling Language (UML), which focuses on the notation and is not a design method. Ivar Jacobson later joined Rational and became the third member of the triad that developed the UML.

The UML attempts to be a unifying notation that incorporates the best of a number of other notations as well as current best practice in one generally applicable notation. That is, you should equally be able to apply the UML to a real-time system, a payroll system or a Web browser. Each project might make more or less use of different parts of the UML (and indeed some parts may be ignored by different projects). However, the UML should act as a common vocabulary for all object-oriented design projects. Possibly surprisingly, this is what has begun to happen. Almost all (if not all) object-oriented design tools now support

the UML (often in addition to their own notation), and many books have been written on how to apply the UML in different situations (in some cases with additions being made).

One of the most significant aspects of the UML is that it possesses a meta-model. This is a model which explicitly describes the UML (in fact this meta-model is written in UML!) thus allowing different tool vendors to implement the UML with the same meaning. It also allows different tool vendors to exchange models if they wish. It also provides a concrete basis upon which others can assess, review and respond to the UML. This was a very significant development when the UML was first released, and provided a very firm foundation for the UML as the notation of choice.

3.2.1 History of the UML

The UML was not developed overnight (see Figure 3.1). It has gone through an extensive development process which started in the mid-1990s. As stated earlier, my first encounter with what was to become the UML was when it was first documented as part of the Unified Method (release 0.8) in October 1995. At this point in time its heaviest influences were OMT (where I was coming from) and the Booch method. This was primarily because the two key architects at this time were Rumbaugh and Booch. However, OMT has had many influences and has taken many elements from other design methods (see Figure 3.2).

At this time the Unified Method was an impressive exercise, as it had only been under development for the best part of a year. However, things did not stand still, and by the middle of the next year (1996) version 0.9 was released (and version 0.91 three months later). The name had at this point changed and the release was now called the Unified

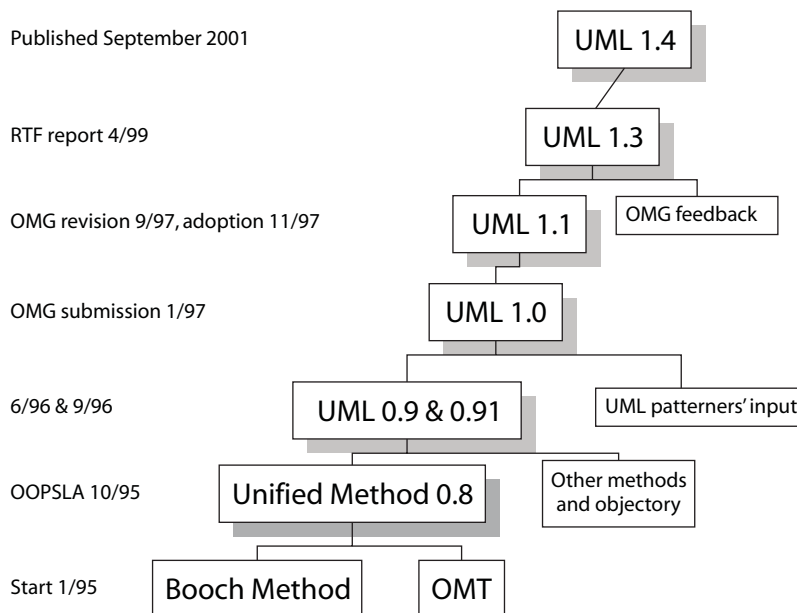


Figure 3.1 A potted history of the UML.

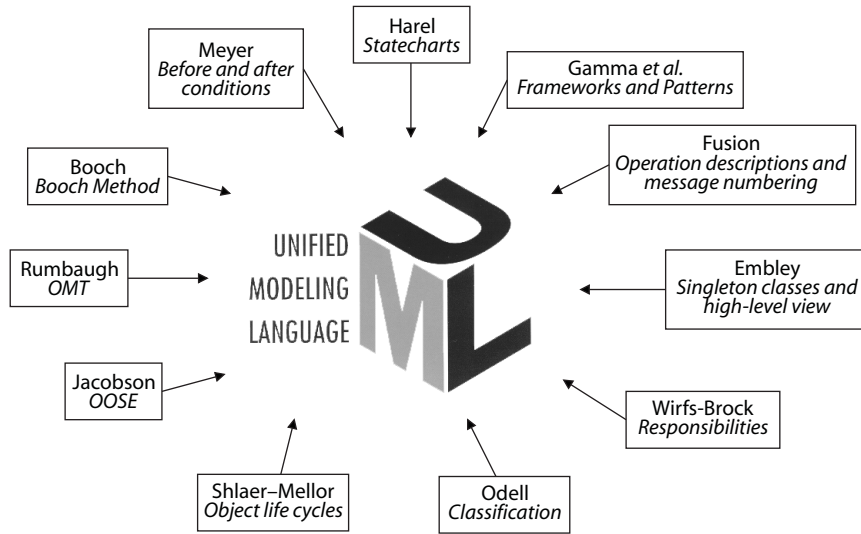


Figure 3.2 Influences on the UML.

Modeling Language. This release focused on the notation and mostly ignored the process. However, much had happened during this time. Not only had many people worldwide commented on the 0.8 release (as it was freely available for download from Rational's Web site – <http://www.rational.com/>), but the influence of Ivar Jacobson was now being felt. Jacobson had been one of the key architects of the Objectory method, which was most notable for its use of use cases. He had joined Booch and Rumbaugh at Rational at just about the same time as the 0.8 version was released. In UML 0.9 use cases were seen for the first time.

Other partners were now becoming involved in the UML development process, ensuring that a wide variety of backgrounds, expertise and experience was brought to bear. Companies such as IBM, Hewlett-Packard and Microsoft all contributed. Then, at the beginning of 1997, the UML 1.0 standard was presented to the OMG for acceptance as an OMG standard. Version 1.1 of the UML was promulgated as a standard by the OMG towards the end of 1997.

Development of the UML has not stood still, however, although it is now under the control of the OMG. Rather it has continued to develop, and we are now at version 1.4 of the UML (September 2001). At the time of writing there were at least three initiatives looking at developments to the UML, including UML 1.4 with Action Semantics, which adds to UML the syntax and semantics of executable actions and procedures, including their run-time semantics. These semantics are contained within one package, labelled Actions, which defines the various kinds of actions that may compose a procedure. In addition the RTF has been published for UML 1.4.1.

UML is now almost universal as the language used to describe object-oriented models. Indeed, incorporating UML as an OMG standard has ensured that many organizations have adopted it as a non-proprietary standard and that the standard has maintained pace with current developments in computer science – the Internet and Java in particular.

3.2.2 Introduction to the UML

The UML is made of a number of models that together describe the system being designed. Each model comprises one or more diagrams with supporting documentation and descriptions (note that the diagrams alone are not enough). Each model is intended as a complete description of the system being designed from a particular perspective (for example, the static structure of the classes or the dynamic behaviour of the operating system). Each diagram can be part of more than one model, or different parts of the same diagram can be part of different models.

The primary diagrams that comprise the UML are listed below and presented in Figure 3.3.

- *Use case diagrams*. Essentially, these present the interactions between users (human or otherwise) and the system. They therefore also highlight the primary functionality of the system.
- *Class diagrams*. These diagrams present the static (class) structure of the system. They are the core of the UML notation and of an object-oriented design.
- *Object diagrams*. These diagrams use notation which is almost identical to class diagrams, but they present the objects and their relationships at a particular point in time. Object diagrams are not as important as class diagrams, but can be very useful.
- *Activity diagrams*. These describe the flow of activities or tasks, typically within an operation. They are a bit like a graphical pseudocode.
- *Sequence diagrams*. These diagrams show the sequence of messages sent between collaborating objects for a particular task. They highlight the flow of control between the objects.
- *Collaboration diagrams*. These diagrams show the sequence of messages sent between collaborating objects for a particular task. The diagrams highlight the relationships between the collaborating objects. Tools such as Rational Rose allow you to generate collaboration diagrams from sequence diagrams (and vice versa).

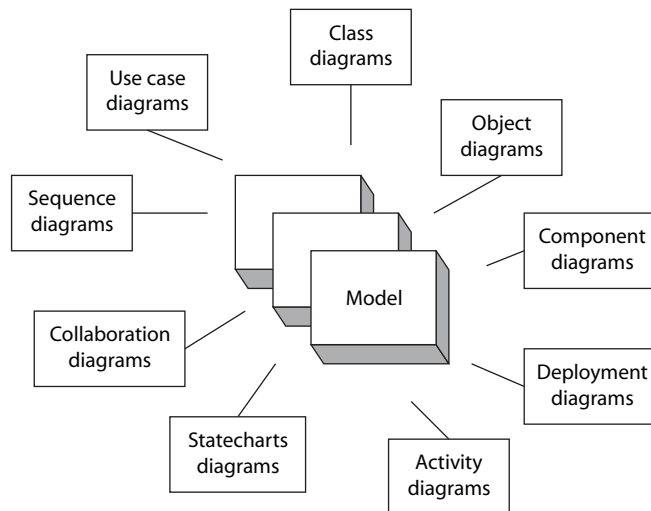


Figure 3.3 Relationship between diagrams and models.

- *Statecharts*. A statechart, or state diagram, illustrates the states an object can be in and the transitions which move the object between states.
- *Component diagrams*. These diagrams are used to illustrate the physical structure of the code in terms of the source code. In Java this means the class files and Java Archive Files (JAR), as well as items such as Web Archive Files (WAR) and Enterprise Archive Files (EAR) in the Java 2 Enterprise Edition architecture.
- *Deployment diagrams*. Deployment diagrams illustrate the physical architecture of the system in terms of processes, networks and the location of components.

3.2.3 Models and Diagrams

UML diagrams are comprised of model elements (a complete listing of which is provided in Appendix A). A diagram represents a particular view into the model. This is because modelling a software system is a complex process in its own right. Ideally the whole system could be described in a single, easily comprehensible diagram. However, only the simplest systems might achieve such an aim. Instead, a model of a software system will only ever be that: a model of the real thing. Models abstract or hide some of the details of the real thing. In the same way, a UML model presents a particular view of a system, abstracting or hiding particular details. Even so, a single model is a complex entity in its own right and is likely to be difficult to present meaningfully within a single diagram. Instead, a particular model can be viewed in different ways. Each UML diagram is just such a view.

One of the key aspects of the UML is that each diagram should be consistent with any other diagrams representing the same information. That is, if one operation is mentioned in two diagrams it should have the same name, with the same return type and the same parameters in each.

When a design is documented using the UML, multiple models are created. Each model captures a different aspect of the emerging design. These aspects are documented in terms of the diagrams, additional notes and documentation. The key element of each model is the visual aspect of the design; however, this visual aspect is augmented by additional textual descriptions and specifications. Indeed, within many diagrams detailed specifications as well as adornments and notes may be included.

The UML is thus a language for:

- visualizing,
- specifying,
- describing, and
- documenting a software system.

However, the UML is not a design method, it is purely a notation for documenting a design (note that the above all relate to describing the design). A notation on its own is not enough: a method indicating how to apply that design is required. Conceptually the UML can be used with any appropriate object-oriented design method. In this book we use it with the Unified Process, the design method developed by Jacobson, Booch and Rumbaugh as a complement to the UML. This design method is introduced in the remainder of this chapter.

3.3 Analysis of the UML

First, it is worth trying to dispel one of the major myths surrounding the UML – it is *not* a methodology or a process! The UML is a notation that can be used to describe object-oriented systems. This notation tells you nothing about how you should go about carrying out a design, producing the elements of a UML diagram or identifying how you should structure your models. Indeed, it is for this very reason that Rational produced the Unified Process as one methodology that can use the UML. Note that I have just said “one methodology”, as the UML can be used with any appropriate methodology and is not directly tied to the Unified Process.

It is important to address the question of why so many people seem to think that UML must be a method or that it is inherently tied to the Unified Process. I believe that there are a number of reasons, including:

1. Originally, notations and methods were very tightly coupled (for example the OMT, OOD, BOOCH etc. all had their own notations, as well as methods). Indeed this was a motivation behind the UML: to try to get away from the “my notation is best” attitude and to overcome the notation wars!
2. Confusion between the UML and the Unified Process. An understandable confusion exists between the UML notation and the method that has been developed (with a similar name) that uses the UML. Of course, this is made worse because both originate from the Unified Method work of the late 1990s.
3. The bandwagon effect. By this I mean that people tend to jump on the UML bandwagon without necessary understanding what UML is, and make assumptions that fit their own knowledge and experience. Hence the number of job adverts that include “must have knowledge and experience of the UML method”.

Another myth to deal with is that the UML is complete. It is not! You will almost always find things that cannot easily be described in UML or that appear to be missing (for example, there is nothing in UML to support describing the data model that your object-oriented application (be it Java, C#, C++ etc). might be working with. In such situations you must resort to using another notation in conjunction with UML. If you are using a UML diagramming tool, this may mean that you must also switch to a different diagramming system.

Finally, UML is actually quite large, and it can take time to get familiar with each type of diagram in UML and how you use them. However, in the main you do not need all of UML for a particular project. For example, there are very many classes in Java. In the last Java software system you built, how many of those packages (let alone classes) did you use? I would be surprised if it was not a core subset that, in general, you use on most projects. It is the same with UML. You should therefore get familiar and comfortable with the appropriate subset that you will need to use for the types of projects you are likely to undertake.

3.4 The Unified Process

The Unified Process is a design framework which guides the tasks, people and products of the design process. It is a framework because it provides the inputs and outputs of each activity, but

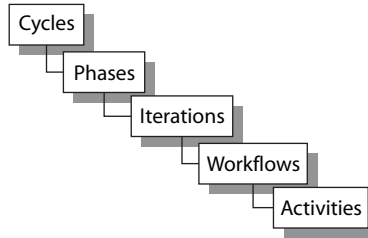


Figure 3.4 Key building blocks of the Unified Process.

does not restrict how each activity must be performed. Different activities can be used in different situations, some being left out, others being replaced or augmented (this is discussed in more detail later in this book). Why then is the Unified Process called a process and not the Unified Framework? It is called a process because its primary aim is to define:

- Who is doing what.
- When they do it.
- How to reach a certain goal (i.e. each activity).
- The inputs and outputs of each activity.

It is thus an engineered process. In fact, it is comprised of a number of different hierarchical elements (see Figure 3.4).

The Unified Process actually comprises low-level activities (such as finding classes), which are combined together into disciplines (formerly known as workflows) which describe how one activity feeds into another). These disciplines are organized into iterations. Each iteration identifies some aspect of the system to be considered. How this is done is considered in more detail later. Iterations themselves are organized into phases. Phases focus on different aspects of the design process, for example requirements, analysis, design and implementation. In turn phases can be grouped into cycles. Cycles focus on the generation of successive releases of a system (for example, version 1.0, version 1.1 etc.).

3.4.1 Overview of the Unified Process

There are four key elements to the philosophy behind the Unified Process. These four elements:

- are iterative and incremental
- are use case-driven
- are architecture-centric
- acknowledge risk

Iterative and Incremental

The Unified Process is iterative and incremental, as it does not try to complete the whole design task in one go. One of the features of the waterfall model of software engineering used by many design methods (see Figure 3.5) is that it primarily assumes that you will complete the require-

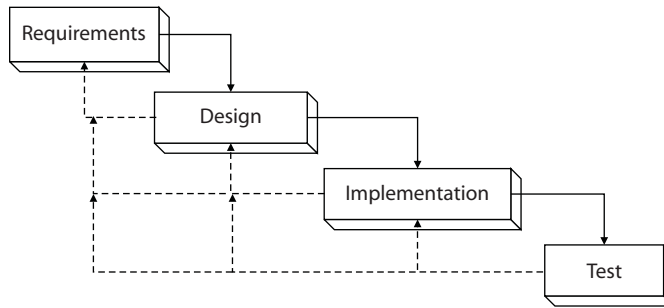


Figure 3.5 The waterfall model.

ments analysis before you start the design phase. In turn, you will complete the design phase before you start the implementation phase and so on. It does accept that there may be some feedback of information from one phase to any preceding phases and that this feedback may have an impact on the products of the preceding phases. However, this is a secondary issue and the assumption is that you will be able to complete the vast majority of one phase before ever considering the next phase. This may be true if this is the fifth or sixth system you have built in the same domain for the same type of application. It is unlikely to be the case with your first application in a new domain (such as your first e-commerce project!).

In contrast to the waterfall model, the Unified Process has an iterative and incremental model. That is, the design process is based on iterations which either address different aspects of the design process or move the design forward in some way (this is the incremental aspect of the model). This does not mean that the Unified Process is a process based on rapid prototyping. Any prototypes that are developed in the Unified Process are used to explore some aspect of the design. This could be to verify some architectural issue for which the design options are similar. Indeed, the use of an iterative and incremental approach in the Unified Process requires more planning (rather than less planning) compared with approaches such as those based on the waterfall model.

Essentially the following holds with the iterative approach in the Unified Process:

- You plan a little.
- You specify, design and implement a little.
- You integrate, test and run.
- You obtain feedback before next iteration.

The end result is that you incrementally produce the system being designed. While you do this you explicitly identify the risks to your design/system up front and deal with them early on (see later). Notice that this does not mean that you are hacking the system together; nor are you carrying out some form of rapid prototyping. However, it does mean that a great deal of planning is required, both initially and as the design develops.

Use Case-driven

The Unified Process is also use case-driven. Remember from earlier that use cases help to identify who uses the system and what they need to do with the system (i.e. the top-level functionality).

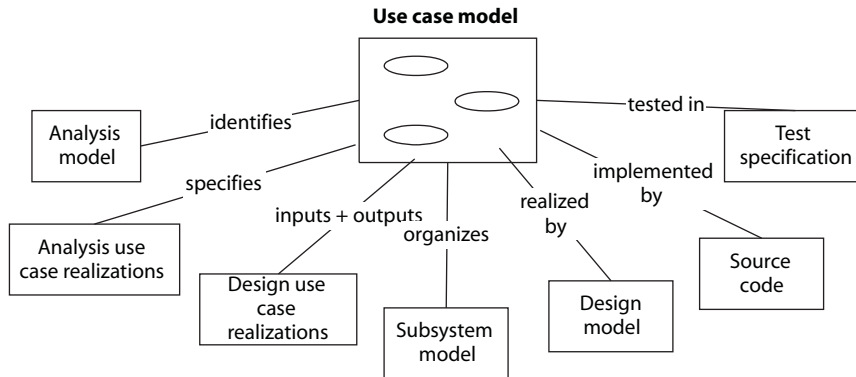


Figure 3.6 The role of use cases.

Thus use cases help identify the primary requirements of the system. One problem with many traditional approaches is that once the requirements have been identified there is no traceability of those requirements through the design to the implementation. Instead, designers (and possibly implementers) must refer back implicitly to the requirements specification and make sure that they have done what is required of them. This is then verified by testing (by which time it is often too late to make any major modifications if the functionality is either wrong or missing).

In the Unified Process use cases are used to ensure that the evolving design is always relevant to what the user required. Indeed, the use cases act as the one consistent thread throughout the whole of the development process, as illustrated in Figure 3.6. For example, at the beginning of the design phase one of the two primary inputs to this phase is the use case model. Then, explicitly within the design model, there are use case realizations which illustrate how each use case is supported by the design. Any use case which does not have a use case realization is not currently supported by the design (in turn, any design elements which do not in some way partake in a use case realization do not support the required functionality of the system!).

To summarize the role of use cases they:

- identify the users of the system and their requirements
- aid in the creation and validation of the system's architecture
- help produce the definition of test cases and procedures
- direct the planning of iterations
- drive the creation of user documentation
- direct the deployment of the system
- synchronize the content of different models
- drive traceability throughout models

Architecture-Centric

One problem with having an iterative and incremental approach is that while one group may be working on part of the implementation another group may be working on part of the design. To ensure that all the various parts fit together there needs to be something. That something is an

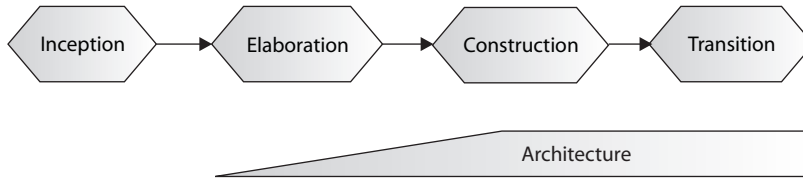


Figure 3.7 The development of the architecture.

architecture. An architecture is the skeleton on which the muscles (functionality) and skin (the user interface) of the system will be hung. A good architecture will be resilient to change and to the evolving design. The Unified Process explicitly acknowledges the need for this architecture by being architecture-centric. It describes how you identify what should be part of the architecture and how you go about designing and implementing the architecture (Figure 3.7). The remainder of the Unified Process then refers back to that architecture.

Obviously, the generation of this architecture is both critical and very hard. Therefore the Unified Process prescribes the successive refinement of the executable architecture, thereby attempting to ensure that the architecture remains relevant.

Acknowledges Risk

Finally, the Unified Process explicitly acknowledges the risk inherent in software design and development. It does this by highlighting unknown aspects in the system being designed and other areas of concern. These areas are then targeted as either being critical to the system and therefore part of the architecture, or areas of risk which need to be addressed early on in the design process (when there is more time) rather than later on (when time tends to be short). Thus it tries to force the riskiest aspects of the system to be designed and implemented early on, hence ensuring that the risk in the system is addressed and managed in a professional manner. Note that it is typically the areas of a design which we do not really understand which end up having the biggest impact on an architecture or the final system. This is often because we do not realize the impact that such areas will have and therefore do not take into account how to deal with their requirements. This is why late on in projects, when such areas are addressed, the system either needs to leave out that functionality or requires major modifications to incorporate the functionality.

Additional Features of the Unified Process

There are two additional features of the Unified Process which are worth making explicit at this stage. The first is that it really requires tool support. That is, it requires a tool that not only supports an appropriate notation (such as the UML; indeed we will assume the UML is the notation used with the Unified Process from now on), but actually supports the Unified Process itself: i.e. a tool which guides you through the various phases, disciplines and activities of the Unified Process. Such a tool can greatly simplify the design process and provide essentially cross-checks and additional support.

The second aspect to note about the Unified Process is that it actually covers the whole of the software development life cycle. That is, it starts by considering the development of the

business case for a software system. It then ends with the long-term ongoing maintenance of a large long-lived software system (the sort of system which is still running after 20 or 30 years, as has been the case with a significant number of COBOL systems which were affected by the millennium bug!).

In this book we do not cover the whole of the software development life cycle from initial concept onwards. Instead, we focus on the stages between the start of the use case analysis (requirements capture) through to the end of the implementation of the software with a brief discussion of the testing phase. We also focus on the four phases which comprise the Unified Process and leave a discussion of the role of cycles until near the end of the book.

3.4.2 Life Cycle Phases

The Unified Process is composed of four distinct phases. These four phases (presented in Figure 3.8) focus on different aspects of the design process. The four phases are Inception, Elaboration, Construction and Transition.

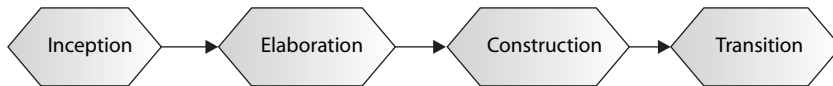


Figure 3.8 The four phases of the Unified Process.

The four phases and their roles are outlined below:

- *Inception.* This phase defines the scope of the project and develops the business case for the system. It also establishes the feasibility of the system to be built. Various prototypes may be developed during this phase to ensure the feasibility of the proposal. Note that we do not focus on the development of the business case in this book: it is assumed that the system to be designed is required and that a business case has already been made.
- *Elaboration.* This phase captures the functional requirements of the system. It should also specify any non-functional requirements to ensure that they are taken into account. The other primary task for this phase is the creation of the architecture to be used throughout the remainder of the Unified Process.
- *Construction.* This phase concentrates on completing the analysis of the system, performing the majority of the design and the implementation of the system. That is, it essentially builds the product.
- *Transition.* The transition phase moves the system into the user's environment. This involves activities such as deploying the system and maintaining it.

Each phase has a set of major milestones that are used to judge the progress of the overall Unified Process (of course, with each phase there are numerous minor milestones to be achieved). The primary milestones (or products) of the four phases are illustrated in Figure 3.9.

A milestone is the culmination of a phase and comprises a set of artefacts (such as specific models) which are the product of the disciplines (and thus activities) in that phase. The primary milestones for each phase are:

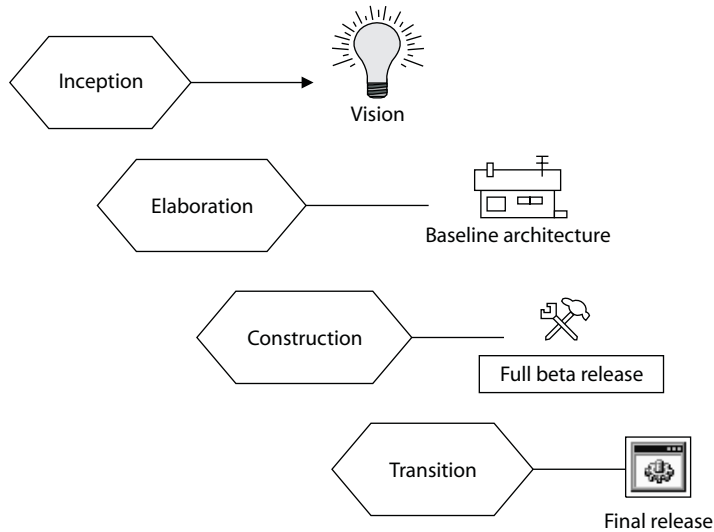


Figure 3.9 The major deliverables of each phase.

- *Inception.* The output of this phase is the vision for the system. This includes a very simplified use case model (to identify what the primary functionality of the system is) and a very tentative architecture, and the most important or significant risks are identified and the elaboration phase is planned.
- *Elaboration.* The primary output of this phase is the architecture, along with a detailed use case model and a set of plans for the construction phase.
- *Construction.* The end result of this phase is the implemented product which includes the software as well as the design and associated models. The product may not be without defects, as some further work has yet to be completed in the transition phase.
- *Transition.* The transition phase is the last phase of a cycle. The major milestone met by this phase is the final production-quality release of the system.

3.4.3 Phases, Iterations and Disciplines

There can be confusion over the relationship between phases and workflows, not least because a single discipline can cross (or be involved in) more than one phase (see Figure 3.10). One way to view the relationships is that the disciplines are the steps you actually follow. However, at different times we can identify different major milestones that should be met. The various phases highlight the satisfaction of these milestones. For example, during the elaboration phase, part of the requirements, analysis, design and even implementation disciplines may be active. However, the emphasis at this time, within these disciplines, will be on elaborating what the system should do and how it should be structured, rather than on the more detailed analysis, design and implementation which occurs during the construction phase.

For the majority of this book we will focus on the various disciplines (not least because this is the emphasis which the designer typically sees). We shall come back to the four phases in Chapter 10.

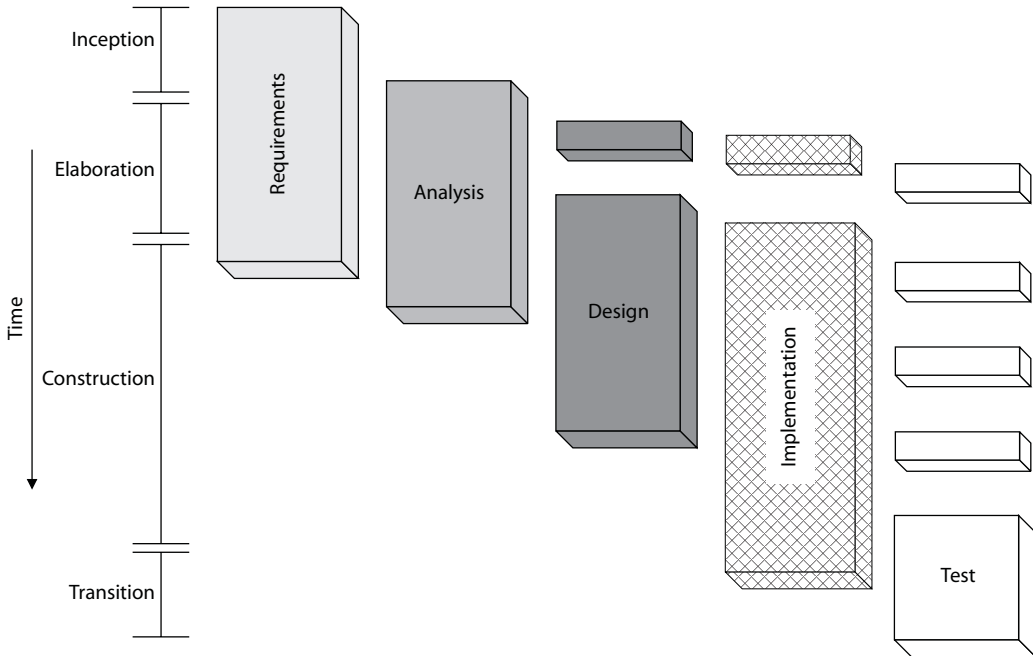


Figure 3.10 Disciplines versus phases.

The five disciplines in the Unified Process are Requirements, Analysis, Design, Implementation and Test (as indicated in Figure 3.10). Note that the Design, Implementation and Test disciplines are broken up. This is to indicate that elements of each discipline may take place earlier than the core parts of the discipline. In particular, the design, implementation and testing of the architecture will happen early on (in the elaboration phase). Thus part of each of the Design, Implementation and Test disciplines must occur at this time.

The focus of each discipline is described below (their primary products are illustrated in Figure 3.11):

- *Requirements*. This discipline focuses on the activities which allow the functional and non-functional requirements of the system to be identified. The primary product of this discipline is the use case model.
- *Analysis*. The aim of this discipline is to restructure the requirements identified in the requirements discipline in terms of the software to be built rather than in the user's less precise terms. It can be seen as a first cut at a design; however, that is to miss the point of what this discipline aims to achieve.
- *Design*. The design discipline produces the detailed design which will be implemented in the next discipline.
- *Implementation*. This discipline represents the coding of the design in an appropriate programming language (for this book that is Java), and the compilation, packaging, deployment and documenting of the software.

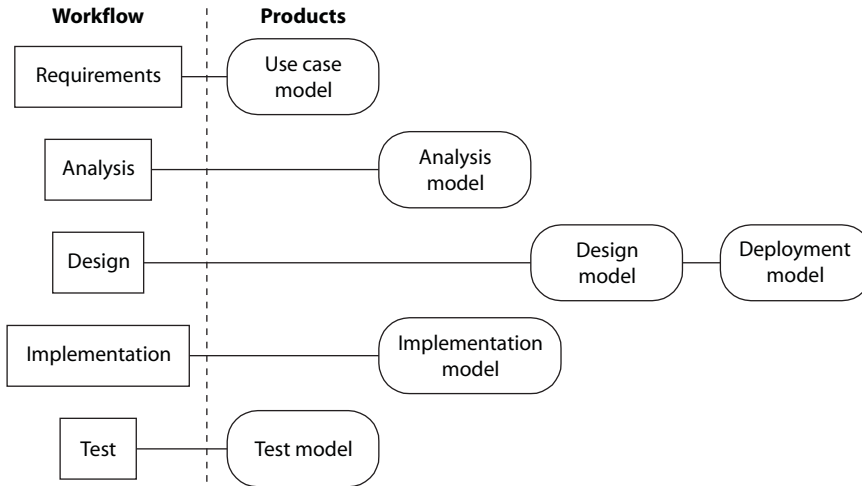


Figure 3.11 Discipline products.

- *Test*. The test discipline describes the activities to be carried out to test the software to ensure that it meets the user's requirements, that it is reliable etc.

Notice that the disciplines all have a period when they are running concurrently. This does not mean that one person is necessarily working on all the disciplines at the same time. Instead, it acknowledges that, in order to clarify some requirement, it may be necessary to design how that requirement might be implemented and even to implement it to confirm that it is feasible.

In fact, this acknowledges that the Unified Process is a spiral (as indicated by its iterative and incremental nature). This is illustrated in Figure 3.12 (note that as a phase moves around the spiral multiple iterations may occur; we have assumed only one iteration in this figure for simplicity's sake). As can be seen from this diagram the five disciplines are

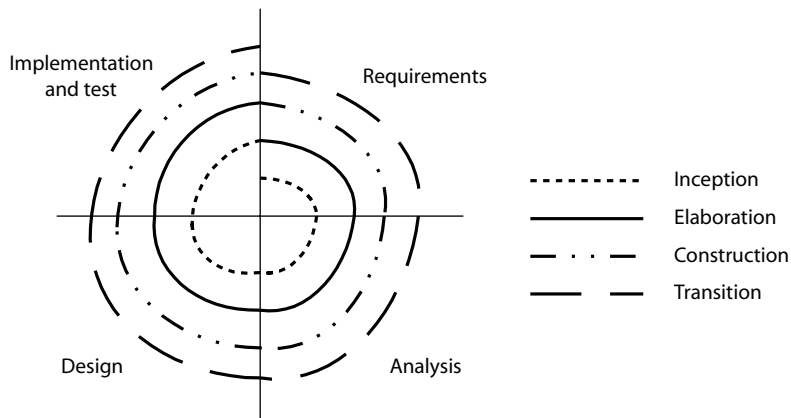


Figure 3.12 The Unified Process is a spiral.

involved in each of the four phases. Each phase moves around the various disciplines producing outputs which feed into the next phase. Each phase examines the requirements (to a greater or lesser extent). Each phase involves the analysis discipline, the design discipline and so on. This is in fact one of the Unified Process's greatest strengths: it represents a practical iterative design method, which is held together by an architecture and which acknowledges risk up front and makes it one of the driving elements of the whole design process. It then ensures that what is being produced will be relevant to users of the system by holding everything together via use cases. Indeed, it is the use cases which help a designer to identify what should be performed in any particular iteration.

3.4.4 Disciplines and Activities

Having discussed disciplines we should mention what disciplines do and what they are comprised of. A discipline describes how a set of activities are related. Activities are the things that actually tell designers what they should be doing. An activity takes inputs and produces outputs. These inputs and outputs are referred to as artefacts. An artefact that acts as an input to a particular activity could be a use case, while the output from that activity could be a class diagram, etc. The actual activities that comprise each of the disciplines will be discussed in more detail in appropriate chapters later in the book; however, Figure 3.13 lists the primary activities for each of the disciplines.

3.4.5 Applying the Unified Process

When it comes to applying the Unified Process to a real-world project, you should notice that it is a framework (see Figure 3.14). This means that there is no universal process which will always be applicable in its entirety. Instead, the Unified Process is designed for flexibility and extensibility. It allows a variety of life cycle strategies and also allows the selection of what artefacts should be produced. It defines what activities should be performed when and which workers should

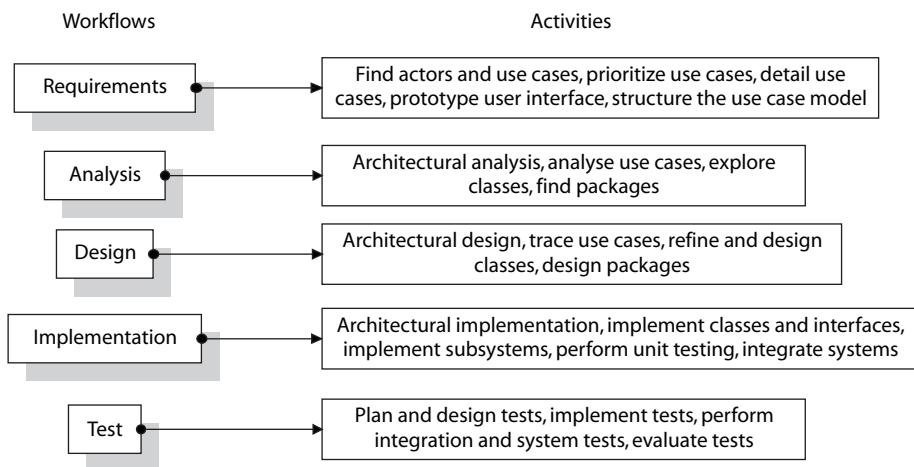


Figure 3.13 Disciplines are comprised of activities.

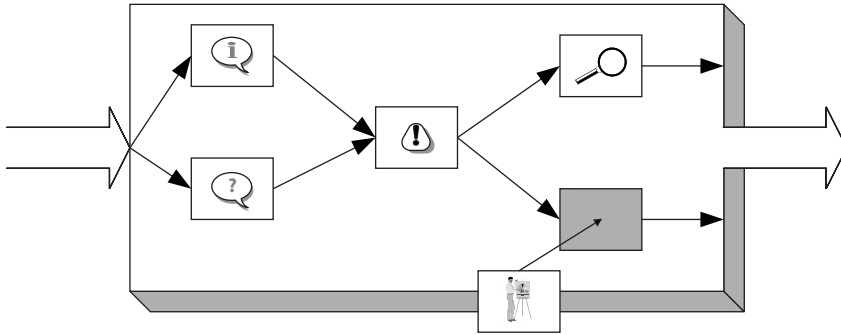


Figure 3.14 The Unified Process is a framework.

perform those activities. Thus it is possible to leave out those elements that don't fit the current project. For example, you might leave out deployment diagrams if you are only deploying on one processor, or if you are working with a batch processing oriented system you may decide to ignore some of the dynamic elements produced, such as statechart diagrams.

In turn, you can add in additional elements if they are required. For example, you may decide to incorporate some real-time extensions into the UML, and some activities to support them. You might decide to incorporate a security view of your system. You might also feel the need to incorporate additional processes. For example, you might incorporate additional activities to help identify an initial set of classes, attributes and relationships. In fact, you may even decide to leave out whole phases, iterations and disciplines as appropriate: for example, a simple system may not need an explicit analysis model!

3.5 The Rational Unified Process

You may have read about the Rational Unified Process (or RUP) elsewhere, so you might think that this is just the Unified Process under another banner (after all, Rational did create the Unified Process in the first place). Well, not exactly. The Rational Unified Process (Kruchten, 2000) is an instantiation of the Unified Process (as is the Enterprise Unified Process or EUP (Ambler, 2001)). Remember that the Unified Process itself is a framework that can be adapted to different situations, types of projects, and detailed techniques. The RUP is a configuration of the Unified Process that is directly supported by the RUP product from Rational. This product specifies many processes and describes how to apply other Rational tools on software projects. It also provides the framework for various documents and artefacts that need to be produced and generally controls the flow of work through a project. This is great if you want to follow this particular instantiation of the Unified Process (but of course it comes at a price). The end result can be that the tool and the process take over! However, it should be noted that the RUP product now has a tool, RUP Builder, for selecting and deploying specialized RUP configurations.

3.6 Summary

To conclude, the Unified Process is a design process framework that is hierarchical, as it is made up of cycles, comprising phases which are themselves made up of workflows that describe how activities are linked. It is engineered because it specifies these activities, who should carry them out (although we don't explicitly identify particular roles for those carrying out activities in this book) and the artefacts produced by the activities. Finally, the key elements of the Unified Process are that it is:

- iterative and incremental
- risk-driven
- architecture-centric
- use case-driven.

3.7 References

Kruchten, P. (2000). *The Rational Unified Process: an Introduction*, 2nd edn. Addison Wesley Longman, Reading, MA.

3.8 Online References

Ambler, S. W. (2001). Enterprise Unified Process White Paper: <http://www.ronin-intl.com/publications/unifiedProcess.htm>
Rational Unified Process: <http://www.rational.com/products/rup/>
OMG Unified Modeling Language: <http://www.omg.org/technology/documents/formal/uml.htm>

Guide to the Unified Process featuring UML, Java and
Design Patterns

Hunt, J.

2003, XVIII, 424 p., Hardcover

ISBN: 978-1-85233-721-6