



## chapter 2

# What is the Unified Process?

### 2.1 Chapter roadmap

This chapter gives a concise overview of the Unified Process (UP). Beginners should start by learning about UP history. If you already know this, then you may choose to skip ahead to Section 2.4, a discussion of UP and the Rational Unified Process (RUP), or to Section 2.5, which discusses how you can apply UP on your project.

Our interest in UP, as far as this book is concerned, is to provide a process framework within which the techniques of OO analysis and design can be presented. You will find a complete discussion of UP in [Jacobson 1] and excellent discussions of the related RUP in [Kroll 1], [Kruchten 2], and also in [Ambler 1], [Ambler 2], and [Ambler 3].

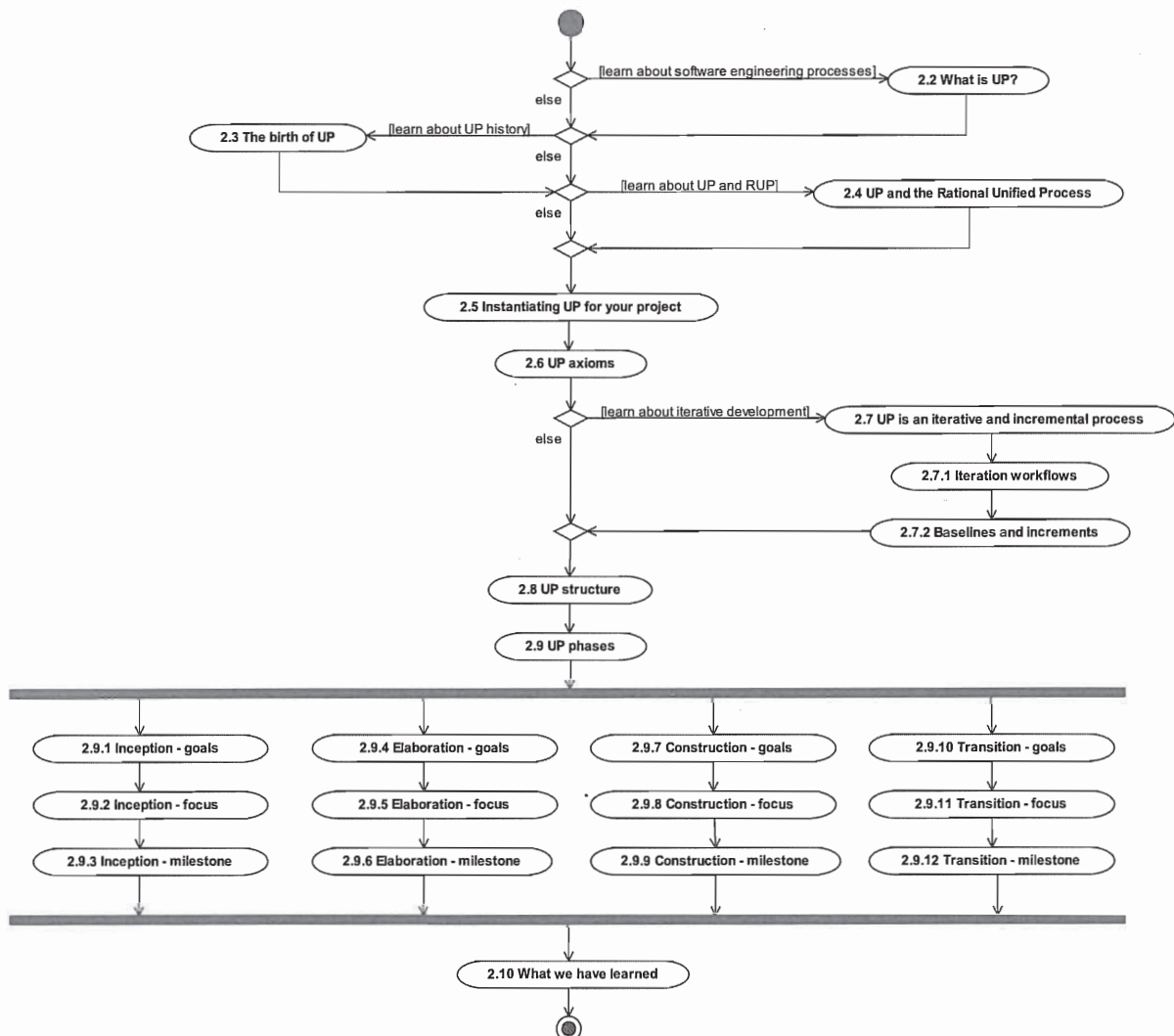


Figure 2.1

## 2.2 What is UP?

A software engineering process (SEP), also known as a software development process, defines the *who*, *what*, *when*, and *how* of developing software. As illustrated in Figure 2.2, a SEP is the process in which we turn user requirements into software.

The Unified Software Development Process (USDP) is a SEP from the authors of the UML. It is commonly referred to as the Unified Process or UP [Jacobson 1]. We use the term UP throughout this book.



Figure 2.2

The UML project was meant to provide both a visual language *and* a software engineering process. What we know today as UML is the visual language part of the project—UP is the process part. However, it's worth pointing out that whereas the UML has been standardized by the OMG, the UP has not. There is therefore still no *standard* SEP to complement UML.

UP is based on process work conducted at Ericsson (the Ericsson approach, 1967), at Rational (the Rational Objectory Process, 1996 to 1997) and other sources of best practice. As such, it is a pragmatic and tested method for developing software that incorporates best practice from its predecessors.

A software engineering process describes how requirements are turned into software.

## 2.3 The birth of UP

When we look at the history of UP, depicted in Figure 2.3, it is fair to say that its development is intimately tied to the career of one man, Ivar Jacobson. In fact, Jacobson is often thought of as being the father of UP. This is not to minimize the work of all of the other individuals (especially Booch) who have contributed to the development of UP; rather, it is to emphasize Jacobson's pivotal contribution.

UP goes back to 1967 and the Ericsson approach, which took the radical step of modeling a complex system as a set of interconnected blocks. Small blocks were interconnected to form larger blocks building up to a complete system. The basis of this approach was “divide and conquer” and it was the forerunner of what is known today as component-based development.

Although a complete system might be incomprehensible to any individual who approaches it as a monolith, when broken down into smaller blocks it can be made sense of by understanding the services each block offers (the interface to the component in modern terminology) and how these blocks fit together. In the language of UML, large blocks are called subsystems, and each subsystem is implemented in terms of smaller blocks called components.

SEP work that was to develop into the UP began in 1967 at Ericsson.

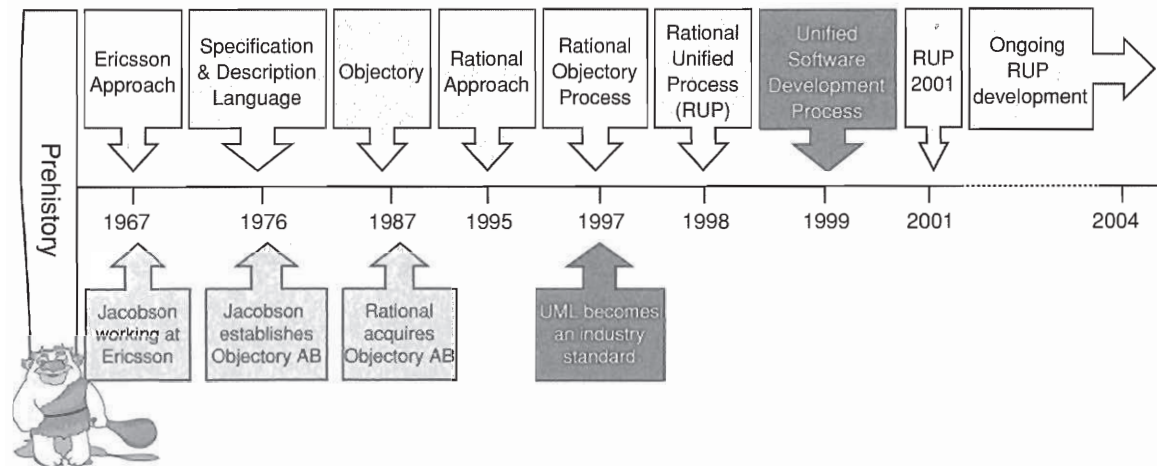


Figure 2.3

Another Ericsson innovation was a way of identifying these blocks by creating “traffic cases” that described how the system was to be used. These traffic cases have evolved over time and are now called use cases in UML. The result of this process was an architecture representation that described all the blocks and how they fitted together. This was the forerunner of the UML static model.

As well as the requirements view (the traffic cases) and the static view (the architecture description), Ericsson had a dynamic view that described how all the blocks communicated with each other over time. This consisted of sequence, communication, and state machine diagrams, all of which are still found in UML, albeit in a much-refined form.

The next major development in OO software engineering was in 1980 with the release of the Specification and Description Language (SDL) from the international standards body CCITT. SDL was one of the first object-based visual modeling languages, and in 1992 it was extended to become object-oriented with classes and inheritance. This language was designed to capture the behavior of telecommunications systems. Systems were modeled as a set of blocks that communicated by sending signals to each other. SDL-92 was the first widely accepted object modeling standard and it is still used today.

In 1987, Jacobson founded Objectory AB in Stockholm. This company developed and sold a software engineering process, based on the Ericsson Approach, called Objectory (*Object Factory*). The Objectory SEP consisted of a set of documentation, a rather idiosyncratic tool, and some probably much needed consultancy from Objectory AB.

Perhaps the most important innovation during this time was that the Objectory SEP was viewed as a system in its own right. The workflows of the process (requirements, analysis, design, implementation, and test) were expressed in a set of diagrams. In other words, the Objectory process was modeled and developed just like a software system. This paved the way for the future development of the process. Objectory, like UP, was also a process framework and needed vigorous customization before it could be applied to any specific project. The Objectory process product came with some templates for various types of software development project, but it almost invariably needed to be heavily customized further. Jacobson recognized that all software development projects are different, and so a “one size fits all” SEP was not really feasible or desirable.

When Rational acquired Objectory AB in 1995, Jacobson went to work unifying the Objectory process with the large amount of process-related work that had already been done at Rational. A 4+1 view of architecture based around four distinct views (logical, process, physical, and development) plus a unifying use case view was developed. This still forms the basis of the UP approach to system architecture. In addition, iterative development was formalized into a sequence of phases (Inception, Elaboration, Construction, and Transition) that combined the discipline of the waterfall life cycle with the dynamic responsiveness of iterative and incremental development. The main participants in this work were Walker Royce, Rich Reitmann, Grady Booch (inventor of the Booch method), and Philippe Kruchten. In particular, Booch’s experience and strong ideas on architecture were incorporated into the Rational Approach (see [Booch 1] for an excellent discussion of his ideas).

The Rational Objectory Process (ROP) was the result of the unification of Objectory with Rational’s process work. In particular, ROP improved areas where Objectory was weak—requirements other than use cases, implementation, test, project management, deployment, configuration management, and development environment. Risk was introduced as a driver for ROP, and architecture was defined and formalized as an “architecture description” deliverable. During this period Booch, Jacobson, and Rumbaugh were developing UML at Rational. This became the language in which ROP models, and ROP itself, were expressed.

From 1997 onward, Rational acquired many more companies bringing in expertise in requirements capture, configuration management, testing, and so on. This led to the release of the Rational Unified Process (RUP) in 1998. Since then, there have been many releases of RUP, each one consistently better than the previous. See [www.rational.com](http://www.rational.com) and [Kruchten 1] for more details.

In 1999, we saw the publication of an important book, the *Unified Software Development Process* [Jacobson 1], which describes the Unified Process.



UP is a mature, open SEP from the authors of UML.

Whereas RUP is a Rational process product, UP is an open SEP from the authors of UML. Not surprisingly, UP and RUP are closely related. We have chosen to use UP rather than RUP in this book as it is an open SEP, accessible to all, and is not tied to any specific product or vendor.

## 2.4 UP and the Rational Unified Process

RUP is a commercial product that extends UP.

The Rational Unified Process (RUP) is a commercial version of UP from IBM, who took over Rational Corporation in 2003. It supplies all of the standards, tools, and other necessities that are not included in UP and that you would otherwise have to provide for yourself. It also comes with a rich, web-based environment that includes complete process documentation and “tool mentors” for each of the tools.

UP and RUP are much more similar than they are different.

Back in 1999 RUP was pretty much a straight implementation of UP. However, RUP has moved on a lot since then and now extends UP in many important ways. Nowadays, we should view UP as the open, general case and RUP as a specific commercial subclass that both extends and overrides UP features. But RUP and UP still remain much more similar than different. The main differences are those of completeness and detail rather than semantic or ideological differences. The basic workflows of OO analysis and design are sufficiently similar that a description from the UP perspective will be just as useful for RUP users. By choosing to use UP in this book, we make the text suitable for the majority of OO analysts and designers who are not using RUP, and also for the significant and growing minority who are.

Both UP and RUP model the *who*, *when*, and *what* of the software development process, but they do so slightly differently. The latest version of RUP has some terminological and syntactic differences to UP, although semantics of the process elements remain essentially the same.

Figure 2.4 shows how the RUP process icons map to the UP icons we use in this book. Notice that there is a «trace» relationship between the RUP icon and the original UP icon. In UML a «trace» relationship is a special type of dependency between model elements that indicates that the element at the beginning of the «trace» relationship is a historical development of the element pointed to by the arrow. This describes the relationship between UP and RUP model elements perfectly.

To model the “who” of the SEP, UP introduces the concept of the worker. This describes a role played by an individual or team within the project. Each worker may be realized by many individuals or teams, and each individual or team may perform as many different workers. In RUP workers are actually called “roles”, but the semantics remain the same.









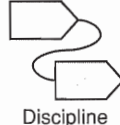

| UP  | RUP   | Semantics   |
|---|---|---|
| <br>Worker   | <br>Role   | Who – A role in the project played by an individual or team                               |
| <br>Activity<br><br>Artifact        | <br>Activity<br><br>Artifact            | What – A unit of work performed by a worker (role) or an artifact produced in the project |
| <br>Workflow<br><br>Workflow Detail | <br>Discipline<br><br>Workflow Detail | When – A sequence of related activities that brings value to the project                  |

Figure 2.4

UP models the “what” as activities and artifacts. Activities are tasks that will be performed by individuals or teams in the project. These individuals or teams will always *adopt specific roles* when they perform certain activities and so, for any activity, UP (and RUP) can tell us the workers (roles) that participate in that activity. Activities may be broken down into finer levels of detail as needed. Artifacts are things that are inputs and outputs to the project—they may be source code, executable programs, standards, documentation, and so on. They can have many different icons depending on what they are, and in Figure 2.4 we show them with a generic document icon.

UP models the “when” as workflows. These are sequences of related activities that are performed by workers. In RUP, high-level workflows, such as Requirements or Test, are given a special name, *disciplines*. Workflows may be broken down into one or more workflow details that describe the activities, roles, and artifacts involved in the workflow. These workflow details are only referred to by name in UP but have been given their own icon in RUP.

## 2.5 Instantiating UP for your project

UP is a generic software development process that has to be instantiated for an organization and then for each particular project. This recognizes that all software projects tend to be different, and that a “one size fits all” approach to SEP just doesn’t work. The instantiation process involves defining and incorporating

UP and RUP must be instantiated for each project.

- in-house standards;
- document templates;
- tools – compilers, configuration management tools, and so on;
- databases – bug tracking, project tracking, and so on;
- life cycle modifications – for example, more sophisticated quality control measures for safety-critical systems.

Details of this customization process are outside the scope of this book but are described in [Rumbaugh 1].

Even though RUP is much more complete than UP, it must still be customized and instantiated in a similar way. However, the amount of work that needs to be done is much less than starting from raw UP. In fact, with *any* software engineering process you can generally expect to invest a certain amount of time and money in instantiation, and you may need to budget for some consultancy from the SEP vendor to help with this.

## 2.6 UP axioms

UP has three basic axioms. It is

- requirements and risk driven;
- architecture-centric;
- iterative and incremental.

We look at use cases in great depth in Chapter 4, but for now let’s just say that they are a way of capturing requirements, so we could accurately say that UP is requirements driven.

UP is a modern SEP that is driven by user requirements and risk.

Risk is the other UP driver because if you don’t actively attack risks they will actively attack you! Anyone who has worked in a software development project will no doubt agree with this sentiment, and UP addresses this by predicating software construction on the analysis of risk. However, this is really a job for the project manager and architect, and so we don’t cover it in any detail in this book.



The UP approach to developing software systems is to develop and evolve a robust system architecture. Architecture describes the strategic aspects of how the system is broken down into components and how those components interact and are deployed on hardware. Clearly, a quality system architecture will lead to a quality system, rather than just an ad hoc collection of source code that has been hacked together with little forethought.

Finally, UP is iterative and incremental. The iterative aspect of UP means that we break the project into small subprojects (the iterations) that deliver system functionality in chunks, or increments, leading to a fully functional system. In other words, we build software by a process of stepwise refinement to our final goal. This is a very different approach to software construction compared to the old waterfall life cycle of analysis, design, and build that occur in a more or less strict sequence. In fact, we return to key UP workflows, such as analysis, several times throughout the course of the project.

## 2.7 UP is an iterative and incremental process

UP aims to build a robust system architecture incrementally.

To understand UP, we need to understand iterations. The idea is fundamentally very simple—history shows that, generally speaking, human beings find small problems easier to solve than large problems. We therefore break a large software development project down into a number of smaller “mini projects”, which are easier to manage and to complete successfully. Each of these “mini projects” is an iteration. The key point is that each iteration contains *all* of the elements of a normal software development project:

- planning
- analysis and design
- construction
- integration and test
- an internal or external release

Each iteration generates a baseline that comprises a *partially complete* version of the final system and any associated project documentation. Baselines build on each other over successive iterations until the final finished system is achieved.

The difference between two consecutive baselines is known as an increment. This is why UP is known as an iterative and incremental life cycle.

As you will see in Section 2.8, iterations are grouped into phases. Phases provide the macrostructure of UP.

### 2.7.1 Iteration workflows

UP has five core workflows.

In each iteration, five core workflows specify what needs to be done and what skills are needed to do it. As well as the five core workflows there will be other workflows such as planning, assessment, and anything else specific to that particular iteration. However, these are not covered in UP.

The five core workflows are

- requirements – capturing what the system should do;
- analysis – refining and structuring the requirements;
- design – realizing the requirements in system architecture;
- implementation – building the software;
- test – verifying that the implementation works as desired.

Some possible workflows for an iteration are illustrated in Figure 2.5. We look at the requirements, analysis, design, and implementation workflows in more detail later in the book (the test workflow is out of scope).

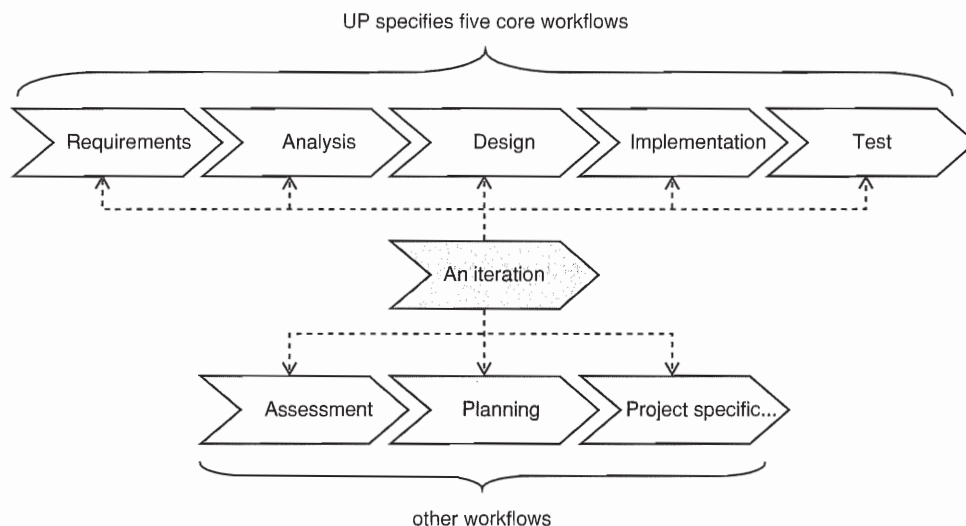


Figure 2.5

Although each iteration may contain all of the five core workflows, the emphasis on a particular workflow depends on where the iteration occurs in the project life cycle.

Breaking the project down into a series of iterations allows a flexible approach to project planning. The simplest approach is just a time-ordered sequence of iterations, where each leads to the next. However, it is often

possible to schedule iterations in parallel. This implies an understanding of the dependencies between the artifacts of each iteration and requires an approach to software development predicated on architecture and modeling. The benefit of parallel iterations is better time-to-market and perhaps better utilization of the team, but careful planning is essential.

### 2.7.2 Baselines and increments

Every UP iteration generates a baseline. This is an internal (or external) release of the set of reviewed and approved artifacts generated by that iteration. Each baseline

- provides an agreed basis for further review and development;
- can be changed *only* through formal procedures of configuration and change management.

Increments, however, are just the *difference* between one baseline and the next. They constitute a step toward the final, delivered system.

## 2.8 UP structure

UP has four phases, each of which ends with a major milestone.

Figure 2.6 shows the structure of UP. The project life cycle is divided into four phases—Inception, Elaboration, Construction, and Transition—each of which ends with a major milestone. Within each phase we can have one or more iterations, and in each iteration we execute the five core workflows and any extra workflows. The exact number of iterations per phase depends on the size of the project, but each iteration should last no more than two to three months. The example is typical for a project that lasts about 18 months and is of medium size.

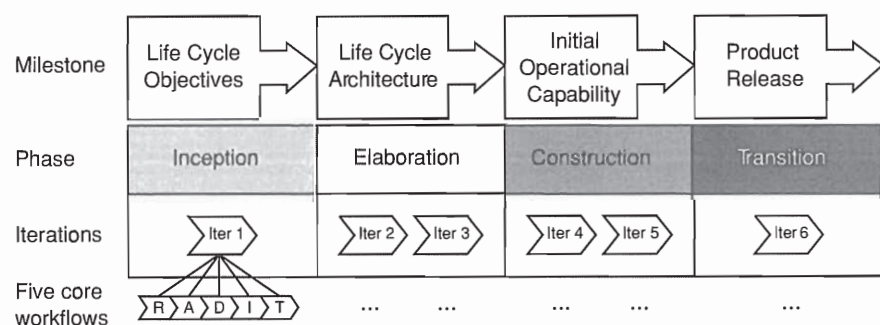


Figure 2.6

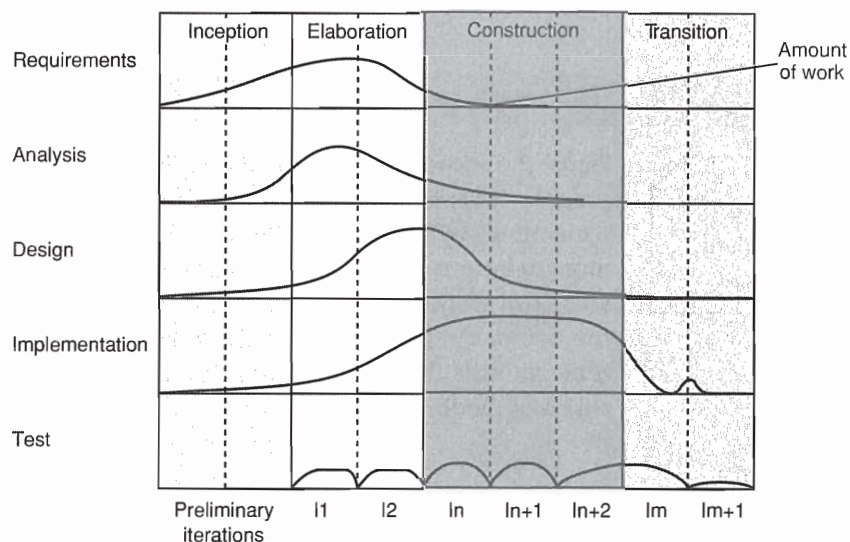
As can be seen from Figure 2.6, UP consists of a sequence of four phases, each of which terminates with a major milestone:

- Inception – Life Cycle Objectives;
- Elaboration – Life Cycle Architecture;
- Construction – Initial Operational Capability;
- Transition – Product Release.

As the project shifts through the phases of the UP, so the amount of work that is done in each of the five core workflows changes.

The amount of work done in each core workflow varies according to the phase.

Figure 2.7 is really the key to understanding how UP works. Along the top, we have the phases. Down the left-hand side, we have the five core workflows. Along the bottom, we have some iterations. The curves show the relative amount of work done in each of the five core workflows as the project progresses through the phases.



**Figure 2.7** Adapted from Figure 1.5 [Jacobson 1] with permission from Addison-Wesley

As Figure 2.7 shows, in the Inception phase most of the work is done in requirements and analysis. In Elaboration the emphasis shifts to requirements, analysis, and some design. In Construction the emphasis is clearly on design and implementation. Finally, in Transition the emphasis is on implementation and test.

UP is a goal-based process rather than a deliverable-based process.

One of the great features of UP is that it is a goal-based process rather than a deliverable-based process. Each phase ends with a milestone that consists of a set of conditions of satisfaction, and these conditions may involve the creation of a particular deliverable, or not, depending on the specific needs of your project.

In the rest of this chapter, we give a brief overview of each of the UP phases.

## 2.9 UP phases

Every phase has a goal, a focus of activity with one or more core workflows emphasized, and a milestone. This will be our framework for investigating the phases.

### 2.9.1 Inception – goals

The goal of Inception is to “get the project off the ground”. Inception involves

Inception is about initiating the project.

- establishing feasibility – this may involve some technical prototyping to validate technology decisions or proof of concept prototyping to validate business requirements;
- creating a business case to demonstrate that the project will deliver quantifiable business benefit;
- capturing essential requirements to help scope the system;
- identifying critical risks.

The primary workers in this phase are the project manager and system architect.

### 2.9.2 Inception – focus

The primary emphasis in Inception is on requirements and analysis workflows. However, some design and implementation might also be done if it is decided to build a technical, or proof-of-concept, prototype. The test workflow is not generally applicable to this phase, as the only software artifacts are prototypes that will be thrown away.

### 2.9.3 Inception – milestone: Life Cycle Objectives

While many SEPs focus on the creation of key artifacts, UP adopts a different approach that is goal-oriented. Each milestone sets certain goals that *must* be



You only create a deliverable when it adds true value to your project.

achieved before the milestone can be considered to have been reached. Some of these goals might be the production of certain artifacts and some might not.

The milestone for Inception is the Life Cycle Objectives. The conditions that must be met for this milestone to be attained are given in Table 2.1. We also suggest a set of deliverables that you may need to create to realize these conditions. However, please remember that you only create a deliverable when it adds true value to your project.

**Table 2.1**

| Conditions of satisfaction  | Deliverable  |
|---|--|
| The stakeholders have agreed on the project objectives                      | A vision document that states the project's main requirements, features, and constraints |
| System scope has been defined and agreed on with the stakeholders           | An initial use case model (only about 10% to 20% complete)                               |
| Key requirements have been captured and agreed on with the stakeholders     | A project glossary   |
| Cost and schedule estimates have been agreed on with the stakeholders       | An initial project plan  |
| A business case has been raised by the project manager                      | A business case  |
| The project manager has performed a risk assessment                         | A risk assessment document or database   |
| Feasibility has been confirmed through technical studies and/or prototyping | One or more throwaway prototypes   |
| An architecture has been outlined   | An initial architecture document   |

#### 2.9.4 Elaboration – goals

The goals of Elaboration may be summarized as follows:

- create an executable architectural baseline;
- refine the risk assessment;
- define quality attributes (defect discovery rates, acceptable defect densities, and so on);

Elaboration is about creating a partial but working version of the system – an executable architectural baseline.

- capture use cases to 80% of the functional requirements (you'll see exactly what this involves in Chapters 3 and 4);
- create a detailed plan for the construction phase;
- formulate a bid that includes resources, time, equipment, staff, and cost.

The main goal of Elaboration is to create an executable architectural baseline. This is a real, executable system that is built according to the specified architecture. It is *not* a prototype (which is throwaway), but rather the “first cut” of the desired system. This initial executable architectural baseline will be added to as the project progresses and will evolve into the final delivered system during the Construction and Transition phases. Because future phases are predicated on the results of Elaboration, this is perhaps the most critical phase. In fact, this book focuses very much on the Elaboration activities.

### 2.9.5 Elaboration – focus

In the Elaboration phase, the focus in each of the core workflows is as follows:

- requirements – refine system scope and requirements;
- analysis – establish what to build;
- design – create a stable architecture;
- implementation – build the architectural baseline;
- test – test the architectural baseline.

The focus in Elaboration is clearly on the requirements, analysis, and design workflows, with implementation becoming very important at the end of the phase when the executable architectural baseline is being produced.

### 2.9.6 Elaboration – milestone: Life Cycle Architecture

The milestone is the Life Cycle Architecture. The conditions of satisfaction for this milestone are summarized in Table 2.2.

### 2.9.7 Construction – goals

Construction evolves the executable architectural baseline into a complete, working system.

The goal of Construction is to complete all requirements, analysis, and design and to evolve the architectural baseline generated in Elaboration into the final system. A key issue in Construction is *maintaining the integrity of the system architecture*. It is quite common once delivery pressure is on and coding begins in earnest for corners to be cut, leading to a corruption of the

Table 2.2

| Conditions of satisfaction  | Deliverable   |
|---|---|
| A resilient, robust executable architectural baseline has been created  | The executable architectural baseline                       |
| The executable architectural baseline demonstrates that important risks have been identified and resolved   | UML static model<br>UML dynamic model<br>UML use case model |
| The vision of the product has stabilized  | Vision document   |
| The risk assessment has been revised  | Updated risk assessment                                     |
| The business case has been revised and agreed with the stakeholders   | Updated business case                                       |
| A project plan has been created in sufficient detail to enable a realistic bid to be formulated for time, money, and resources in the next phases | Updated project plan  |
| The stakeholders agree to the project plan  |   |
| The business case has been verified against the project plan  | Business case   |
| Agreement is reached with the stakeholders to continue the project  | Sign-off document   |

architectural vision and a final system with low-quality and high-maintenance costs. Clearly, this outcome should be avoided.

### 2.9.8 Construction – focus

The emphasis in this phase is on the implementation workflow. Just enough work is done in the other workflows to complete requirements capture, analysis, and design. Testing also becomes more important—as each new increment builds on the last, both unit and integration tests are now needed. We can summarize the kind of work undertaken in each workflow as follows:

- requirements – uncover any requirements that had been missed;
- analysis – finish the analysis model;
- design – finish the design model;
- implementation – build the Initial Operational Capability;
- test – test the Initial Operational Capability.

### 2.9.9 Construction – milestone: Initial Operational Capability

In essence, this milestone is very simple—the software system is finished ready for beta testing at the user site. The conditions of satisfaction for this milestone are given in Table 2.3.

Table 2.3

| Conditions of satisfaction   | Deliverable   |
|--|---|
| The software product is sufficiently stable and of sufficient quality to be deployed in the user community | The software product<br>The UML model<br>Test suite |
| The stakeholders have agreed and are ready for the transition of the software to their environment         | User manuals<br>Description of this release         |
| The actual expenditures vs. the planned expenditures are acceptable  | Project plan  |

### 2.9.10 Transition – goals

The Transition phase starts when beta testing is completed and the system is finally deployed. This involves fixing any defects found in the beta test and preparing for rollout of the software to all the user sites. We can summarize the goals of this phase as follows:

Transition is about deploying the completed system into the user community.

- correct defects;
- prepare the user sites for the new software;
- tailor the software to operate at the user sites;
- modify the software if unforeseen problems arise;
- create user manuals and other documentation;
- provide user consultancy;
- conduct a post-project review.

### 2.9.11 Transition – focus

The emphasis is on the implementation and test workflows. Sufficient design is done to correct any design errors found in beta testing. Hopefully, by this point in the project life cycle, there should be very little work being done in the requirements and analysis workflows. If this is not the case, then the project is in trouble.

- Requirements – not applicable.
- Analysis – not applicable.
- Design – modify the design if problems emerge in beta testing.
- Implementation – tailor the software for the user site and correct problems uncovered in beta testing.
- Test – beta testing and acceptance testing at the user site.

### 2.9.12 Transition – milestone: Product Release

This is the final milestone: beta testing, acceptance testing, and defect repair are finished and the product is released and accepted into the user community. The conditions of satisfaction for this milestone are given in Table 2.4.

**Table 2.4**

| Conditions of satisfaction  | Deliverable                               |
|---|---|
| Beta testing is completed, necessary changes have been made, and the users agree that the system has been successfully deployed | The software product                      |
| The user community is actively using the product  |   |
| Product support strategies have been agreed on with the users and implemented   | User support plan<br>Updated user manuals |

## 2.10 What we have learned

- A software engineering process (SEP) turns user requirements into software by specifying *who* does *what*, *when*.
- The Unified Process (UP) has been in development since 1967. It is a mature, open SEP from the authors of UML.
- Rational Unified Process (RUP) is a commercial extension of UP. It is entirely compatible with UP but is more complete and detailed.
- UP (and RUP) must be instantiated for any specific project by adding in-house standards, etc.
- UP is a modern SEP that is:
  - risk and use case (requirements) driven;
  - architecture centric;
  - iterative and incremental.
- UP software is built in iterations:
  - each iteration is like a “mini project” that delivers a part of the system;
  - iterations build on each other to create the final system.
- Every iteration has five core workflows:
  - requirements – capturing what the system should do;
  - analysis – refining and structuring the requirements;
  - design – realizing the requirements in system architecture (how the system does it);



- implementation – building the software;
- test – verifying that the implementation works as desired.
- UP has four phases, each of which ends with a major milestone:
  - Inception – getting the project off the ground: Life Cycle Objectives;
  - Elaboration – evolving the system architecture: Life Cycle Architecture;
  - Construction – building the software: Initial Operational Capability;
  - Transition – deploying the software into the user environment: Product Release.