Part II: The Lifecycle of a Rational Unified Process Project

Chapter 5. Going Through the Four Phases

Chapter 6. The Inception Phase

Chapter 7. The Elaboration Phase

Chapter 8. The Construction Phase

Chapter 9. The Transition Phase

Chapter 5. Going Through the Four Phases

A RUP development cycle goes through four phases: Inception, Elaboration, Construction, and Transition. This cycle ends with the release of a complete software product. What do you do during each of the various phases of the RUP? What happens during Inception, Elaboration, Construction, and Transition? What are you trying to achieve in each phase? What artifacts are produced? What activities are executed? The next four chapters will give you a sense of the dynamics of a RUP project. Each chapter is dedicated to a phase. But before you dive into these chapters, a few words of caution are necessary.

A Major Misconception

Although the four phases of a RUP project (Inception, Elaboration, Construction, and Transition) are run sequentially, remember at all times that the RUP lifecycle is fundamentally iterative and risk-driven. There is a big misconception that we would like to push aside very early in our discussion: The various phases are not simply a renaming (to sound fancy or different) of the classical phases of a waterfall process. From practitioners making their first acquaintance with the RUP, we have frequently heard, "Oh, I get it! In Inception you do all the requirements, in Elaboration you do the high-level design, in Construction you write the code, and you finish the testing in Transition."

In trying to match the RUP to their current practice, they completely miss the point of iterative development. Yes, in the early weeks or months of a project the emphasis is very likely to be more on requirements and during the final weeks or months to be more on testing and polishing. This change in focus across the lifecycle is precisely what is hinted at by the "humps" on the lifecycle iteration graph (see Figure 1.3); the height of the humps varies across the cycle. But inside each phase, you plan iterations (see how in Chapter 12), and each of these iterations includes many of the software development activities to produce tested code as an internal—and later external—release.
Major Milestones

The purpose of the RUP phases, therefore, is not to partition the activities by type: analysis, coding, and so on (this is what you achieve with the concept of disciplines). Rather, the purpose of a phase is to do just enough of any activity required to meet the objectives of this phase by the time you meet the milestone that concludes it. And what you need to achieve in each phase is mostly driven by risk. In other words, the phases define project states, whereas the states are in turn defined by the particular risks you are mitigating, or the questions you are answering.

- In the Inception phase (see Figure 5.1), you will focus on handling the risks related to the business case: Is this project financially worthy? Is it feasible?

**Figure 5.1. Major Milestones.** The major milestones of the RUP are not expressed in terms of completing certain artifacts or documents, like many other methods, but mostly in terms of risk mitigation and completion of the product.

- In the Elaboration phase, you will focus mostly on the technical risks, examining the architectural risks, and maybe revisiting the scope again, as the requirements become better understood.

- In the Construction phase, you will turn your attention to the "logistical" risks and get the mass of the work done; this is the phase where the project reaches its maximum staffing level.

- In the Transition phase, you will handle the risks associated with the logistics of deploying the product to its user base.

These major milestones are key business-level decision points for the project, where major decisions must be made about the continuation of the project and its scope, funding, strategy, delivery, or schedule.
Also, since the phases are not associated with one kind of role, a RUP project is not executed in a pipeline fashion by having a crew of analysts throw requirements over the wall to a team of designers, who throw a design to a bunch of developers, who pass it to the poor testers to take all the blame. A RUP project is a collaboration among all these people, from business analysts to testers. All the RUP roles are involved throughout most of the cycle (except maybe the very early startup of a brand-new project).

No Fixed Workflows

What remains constant across all RUP projects are these major milestones. What you must achieve by the end of each phase should be every team member's main concern. This does not mean that there is in the RUP a fixed workflow, a fixed "recipe," or a predefined set of activities that must be run in each phase. The RUP offers a palette of possibilities, but the context of your project will determine what you will actually use during the phase to achieve its objectives. From one development cycle to another, or from one project to another, the context will be different, and the risks will be different; so which artifacts you start with, the size of the project, its duration, and the number of involved parties will all play a role in dictating what must actually be executed and which artifacts must be produced or revisited.

Of course, in the RUP there are some common process fragments that are replicated across the lifecycle:

- Activities to start and close a project, a phase, or an iteration, and reviews
- Activities related to the detailed design, coding, testing, and integrating of software
- Activities related to Configuration Management, production of releases, and Change Management

These, however, are ancillary to what you need to achieve in that particular project phase.

The worst situation is the case of a project team trying to run the whole RUP and blindly developing all possible artifacts and executing all activities. By forgetting to tailor the RUP product to suit their context exactly, the team is running a high-overhead project, is overburdened very early, and is at risk of failing. You must streamline the RUP to be as low ceremony as suitable for your project (see Chapters 3 and 10).

No Frozen Artifacts

To simplify planning and to give a feeling of achievement or progress, there is also a temptation to try to complete a RUP artifact (model, document, code) in one shot, within one phase, or even within one single iteration, and to freeze it. "Let's build the requirements, have them 'signed off,' and be done with it." Or "The design is now complete."

There is nothing wrong with doing a perfect job early and not having to revisit an artifact. It is
good that some artifacts become stable as you progress. But the objectives of the phases are not described in terms of \textit{finishing} an artifact, but bringing it to the right level of maturity to be able to make the correct decisions about it. As the project evolves and more is understood about the objectives, as difficulties are discovered, and as external changes are introduced, artifacts have to be revisited, updated, and corrected. Therefore, activities that have already been run have to be re-executed. And having gone much too far in polishing an artifact too early may actually lead to much rework later on.

The Vision and the Business Case will be developed during Inception and hopefully will be stable through the end of the cycle. The requirements are built gradually over Inception and Elaboration and should be complete by the end of the Elaboration phase. The architecture will be designed or selected gradually and should be stable enough by the end of the Elaboration phase. But these artifacts are not sacred and untouchable. You may have to alter the Vision, modify the requirements, or change the architectural design in later phases.

And as we wrote earlier, although all the artifacts described in the RUP could potentially play a role in a project, you do not need to develop \textit{all} the artifacts in your project. Moreover, for a specific kind of artifact, use cases (UCs), for example, you may choose to develop some completely because they are delicate and risky, but not others, which can remain in the form of short descriptions because they are trivial or very similar to existing ones. For example, not all use cases are equally important and critical for the success of the project, and you may decide not to fully develop the description of minor ones.

\section*{Three Types of Projects}

In the next four chapters, to give you a better feel of the activities that can take place in each phase, we will use three different examples of RUP projects:

\begin{itemize}
\item \textbf{G} Project Ganymede, a green-field development of a small application. The \textbf{initial development cycle} of a brand-new application where everything, including the requirements, have to be designed from scratch.

\footnote{"Green-field" development refers to developing a new application. The alternative to green-field development is to develop a new version of an existing application ("brown-field").}

\item \textbf{M} Project Mars, a \textbf{green-field development} of a larger system so that we can articulate the major difference with the first example.

\item \textbf{J} Project Jupiter, an \textbf{evolution cycle} of an existing large application (the "version 2.0"); this is more representative of a large number of RUP projects, which only evolve existing systems and do not create them from scratch.
\end{itemize}

There are many more types of projects; the combinations are infinite, but these three types should suffice to give you an idea about the evolving dynamics of a RUP project through its cycle.
When diving into the next four chapters, remember that the focus of each phase is to achieve some key milestone. These milestones have more to do with mitigating risks and achieving some concrete and objective progress toward the delivery of high-quality software than simply completing a wide range of artifacts and to "tick the box" on some arbitrary, predefined checklist.

Chapter 6. The Inception Phase

In this chapter, we'll provide a basic understanding of what Inception, the first phase of the RUP lifecycle, is all about. Many newcomers to the RUP approach get lost in the rich set of activities and guidelines that it provides and often lose perspective on what they are trying to achieve. The RUP approach is actually quite simple: Make sure you have a clear picture of the objectives for each of the four phases (see Figure 6.1) and imagine what these concrete objectives mean for your situation. Understanding what you want to achieve in a phase will help you apply the RUP approach to your projects more effectively; you will select and perform only those activities that will contribute to achieving the objectives of a particular project.

Figure 6.1. The Inception Phase. Inception is the first phase of the RUP lifecycle; it has a well-defined set of objectives and is concluded by the Lifecycle Objective Milestone. Use these objectives to help you decide which activities to carry out and which artifacts to produce.

As you read this chapter, it is important to remember what we discussed in Chapter 3 about low-ceremony and high-ceremony developments. We describe here an approach to software development that should help you develop better software, and we describe what types of artifacts you need to produce. As you apply these guidelines to your project, you need to decide how formally you want to capture artifacts: This could be done in your head, on a whiteboard, or in a model or document. How formally do you want to review these artifacts? We will try to point out obvious shortcuts you can make if you work in small projects or low-ceremony projects, but in the end, nothing can replace common sense, and you will need to make a judgment call as to what fits your project best. Also remember what we discussed in the previous chapter regarding RUP not having fixed workflows and avoiding frozen artifacts.
Objectives of the Inception Phase

Inception is the first of the four lifecycle phases in the RUP. It is really all about understanding the project scope and objectives and getting enough information to confirm that you should proceed—or perhaps convince you that you shouldn’t. The five basic objectives of the Inception phase are

1. **Understand what to build.** Determine the vision, the scope of the system, and its boundaries, that is, what is inside the system and what is outside. Identify who wants this system and what it is worth to them.

2. **Identify key system functionality.** Decide which use cases (which ways of using the system) are most critical.

3. **Determine at least one possible solution.** Identify at least one candidate architecture.

4. **Understand the costs, schedule, and risks** associated with the project.

5. **Decide what process to follow and what tools to use.**

Note that the numbering of the objectives does not indicate priority, nor that they be addressed in any specific order. Quite the opposite: You will address all the objectives in parallel. In this chapter we will describe how to proceed to achieve each of these five objectives.

Inception and Iterations

Most projects have one iteration in the Inception phase. Some projects may, however, need more than one iteration to accomplish the objectives (see also Chapter 12 on the planning of an iterative project).

Among the reasons for multiple iterations, you find

- The project is large, and it is hard for the project team to grasp the scope of the system.
- The system is unprecedented, and it is difficult to pinpoint what the system should do.
- There are many stakeholders, and stakeholder relations are complex, such as difficult contractual negotiations.
- It is difficult to get the business case right in one shot or to develop an optimal balance between the scope of the project and the required investment. This is often the case when building new commercial applications.
- There are major technical risks that need to be mitigated by a prototype, or you need to build a proof-of-concept before you get buy-in from your sponsors. In particular, you may need to prototype your candidate architecture to get a better understanding of its performance, cost, and other characteristics.

If you have more than one iteration, the first iteration typically focuses primarily on objectives 1–3 (the "what" to build), while the latter iterations focus more on objectives 4 and 5 (the "how" to
For each of our three example projects, you would typically have the following iterations pattern:

- **Project Ganymede**, a small green-field project: Since the application is small, you can typically understand what you need to build in a fairly short time frame. You likely need only one iteration.

- **Project Mars**, a large green-field project: Since the application is more complex and you have never built this type of system before, you need to take some time to get buy-in of what you are trying to achieve from all stakeholders. You probably need two iterations in Inception.

- **Project Jupiter**, a second generation of a large project: You are starting with an existing system, whose characteristics are understood, and you have a very good idea of what needs to be developed in this second generation. You have a number of use cases, extensions of existing ones or new ones, and some nonfunctional requirements, as well as a number of defects of the first system that you know you want to address. You still need to create an inventory of what to do so you do not miss any high-priority enhancements. It should be sufficient to do one iteration, and that iteration can probably be slightly shorter than most other iterations in your project.

---

**Objective 1: Understand What to Build**

This may sound strange, but the fact is that in many projects there is no common understanding of what needs to be built. Although all team members may think they know, often each one has a completely different understanding than the next. If you want to succeed, all stakeholders must agree on a common definition of success. You need to make sure that customers, management, analysts, developers, testers, technical writers, and other key people involved agree on what system to build.

To ensure a common understanding, you need to

A. **Agree on a high-level Vision.**

B. **Provide a "mile-wide, inch-deep" description** of the system. Briefly describe what the system should do, without providing so much detail that you get bogged down or that some stakeholders, such as customers or management, lose sight of what is being built because key information is hidden in a mass of requirements documentation.

C. **Detail key actors and use cases** so that all stakeholders can easily understand them and team members can use them as direct input for their work.
Produce a Vision Document

To address item A, you produce a Vision Document. For very small projects, this could be an informal document, maybe even an e-mail message capturing a previous whiteboard discussion. For average-sized projects, you might write a Vision Document of a few pages. No matter what the format is, a Vision should clarify to stakeholders

- The benefits and opportunities that will be provided by building the application.
- The problem(s) the application will solve.
- Who the target users are.
- At a very high level, what the product will do, expressed as high-level features or in terms of outlining a few key use cases.
- Some of the most essential nonfunctional requirements, such as supported operating systems, database support, required reliability, scalability, and quality, as well as licensing and pricing, if that is relevant.

The Vision creates the foundation for a common understanding of the motivation for building the system, as well as a high-level definition of the system to be built. The Vision should be complete and stable at the end of Inception, but you will continue refining it throughout the project, especially in Elaboration and early Construction (if there is a significant change of scope). It is important that the Vision is public, shared, and constantly reviewed, so that no one can say he or she didn't know or understand it. The widely used Statement of Work (SOW) is somewhat analogous to parts of the Vision.

For more information, see the section Develop a Vision, in Chapter 15.

Generate a "Mile-Wide, Inch-Deep" Description

As item B specifies, you need to provide a good description of the system’s scope without going into too much detail. The description requires two basic activities:

- **Identify and briefly describe actors.** Identify typical users for your system, and classify them based on what they do and what services they expect. Identify also other systems with which yours will interact. Capture these user types and external systems as actors.

- **Identify and briefly describe use cases.** Identify and describe how each actor will interact with the system. If the actor is a human, then describe typical ways the actor will use the system. Descriptions of these typical interactions with the system are called use cases.

At this stage, do not get into too many details: A description a couple of paragraphs long is sufficient for most use cases. However, you could spend a little more time on the use cases you identify as the most critical (and they should not be more than 20 percent of the total number) so that you have a solid understanding of them.

Hold a Workshop or Brainstorming Session
So, how do you produce this mile-wide, inch-deep description? For small projects, you get your team, your customer, and maybe other stakeholders together for a brainstorming meeting of a few hours. For larger projects, you may do a two-day workshop that includes all key stakeholders: project manager, architect, lead developer, customer, and a couple of analysts. During this session, your goal is to complete the seven steps that follow. Note that you do the steps in an iterative fashion, meaning that you will revisit these steps several times in Inception. It is often useful to "time-box" each step, that is, assign a fixed amount of time you will dedicate to it. As you run out of time on a step, go to the next step, and revisit the previous steps at a later stage. This will prevent participants from becoming too involved in one issue, forgetting that you want to achieve breadth more than depth. Note that if you have a brainstorming meeting, you will spend less time on each step, especially step 4. Have people work on step 4 after the meeting, and then have a follow-up meeting where you revisit steps 4–7.

**Step 1.** Identify as many actors as you can (remember, actors represent both users and external systems with which the system interacts). Throughout Inception and Elaboration you will eliminate some actors that are not needed, merge some actors that have similar needs, and add some additional actors you forgot. Write a one-sentence description of actors.

**Step 2.** Associate each actor with use cases, capturing the actor's interactions with the system by providing a brief description of the use case.

**Step 3.** For each of the use cases, determine whether it requires interaction with other users or systems. This will help you identify additional actors. Continue to go back and forth between finding actors and use cases until you think you have identified enough to understand the scope of the system. You most likely have not gotten them all, but it is good enough at this stage.

When you've completed these first three steps, your results will probably look similar to those shown in the use-case diagram in Figure 6.2.

**Step 4.** Write a paragraph describing each actor and a couple of paragraphs about each use case. This can be done as a break-out session in which each person in the workshop is given two hours to describe, for example, one actor and two to three use cases. Make sure there is overlap; that is, the same use case and actors are given to several participants to describe. Then reassemble the group and review all descriptions. At this point you may encounter an interesting phenomenon: Although everyone agrees on the use-case names, each person has his or her own interpretation of what the use case entails. With detailed, written descriptions, these different interpretations become clear; by comparing them, you can see whether you need to come up with a few more use cases to cover all the functionality needed.

**Step 5.** Create a Glossary containing the key "items" the application deals with. For an insurance application, for example, you'd want to define things such as claims, different types of policies, and so on. Also add common terminology that is used in the domain and that the team or stakeholders may not be familiar with. This glossary should help the participants agree on a common terminology as this is often a cause of miscommunication and misunderstanding.

**Step 6.** Review key items that the system is dealing with, and make sure you have created use cases that detail how each key item is created, maintained, and deleted. Do you have a use case that describes how to set up a policy? How to make changes to a policy? How to cancel a policy? This is a great way of discovering holes in your use-case model, and spending only 30 minutes on this activity will often get you a long way.

**Step 7.** At this stage, you need to identify the most essential or critical use cases (maybe only one or two, and at most 20 percent of your use cases). See the section Identify Key
System Functionality for more details.

**Figure 6.2. System Overview: User Kinds and Their Use Cases.** During a use-case workshop, capture (on a whiteboard) the user groups and systems (actors) that will interact with your system and the services (use cases) your system will provide to those actors.

For more information, see the section Describe Requirements "Mile-Wide, Inch-Deep," in Chapter 15.

**Detail Key Actors and Use Cases**

Another step in understanding what you must build is to refine some of the use cases. At the end of the workshop or brainstorming session, you assign one or several use cases to each analyst, who will describe in further detail essential or critical use cases identified in step 7. Typically, you'll generate a couple of pages for each one. The higher the ceremony, the more detailed the description.

For more information on detailing actors and use cases, see the section Detail Actors and Use Cases, in Chapter 15.

In parallel to writing the use-case descriptions, you should also develop user-interface prototypes. This allows you to visualize the flow of events, making sure that you, the users, and other stakeholders understand and agree on the flow of events. See the section Develop User-Interface Prototypes, in Chapter 15, for more information.

You should time-box the activities on use cases to avoid getting bogged down into too much detail. You should also focus on capturing the most essential flow of events and point out the existence of alternative flows of events (because it will help you to assess how complex the use case is to implement), rather than describing the details of each flow of events.

Especially for small projects, you often have the same person take on the role of analyst and developer, meaning that the person who describes a use case will also implement it. If so, you may want to spend less time on documenting the detailed requirements and come back to them as you implement the use case. It is still very useful to identify alternative flows of events, since this will be useful to you when you estimate the amount of work remaining.
For each of our three example projects, the teams do the following:

- **Project Ganymede, a small green-field project:** The project manager/architect spends a day writing a three-page Vision Document. The team spends half a day in a brainstorming session to come up with an initial set of actors and use cases. Thirteen use cases are found, and each team member takes four to five use cases, spending 30 minutes to detail each use case. Then they get together and realize that they need another four use cases. They merge two use cases and eliminate one. They spend another two hours describing each of the four most critical use cases (see the section **Objective 2: Identify Key System Functionality**).

- **Project Mars, a large green-field project:** In the first iteration, the analysts, involving key stakeholders as necessary, spend roughly a week writing a first-draft Vision of eight pages. A lot of time is spent getting buy-in from all stakeholders. The team spends two days on a use-case workshop with eight key stakeholders, allowing them to come up with a good first-draft use-case model and glossary. In the second iteration, the team refines the Vision, which will be further refined later in the project, especially in Elaboration. They spend roughly four hours describing each of the nine most critical use cases (see the following section, **Objective 2: Identify Key System Functionality**). In conjunction with doing that, they make a number of updates to the use-case model.

- **Project Jupiter, a second generation of a large project:** The team makes some updates to the existing Vision, clearly labeling what will be accomplished in the second generation, which is done in a day or two. Most of the time is spent on making an inventory of additional capabilities to implement and the known problems in the first system that must be fixed. The team tries to identify use cases that need to be added, and the most critical of the new use cases are detailed, with the team spending two to four hours on each of them. Planned improvements to existing use cases are listed, but these improvements are normally not detailed at this stage.

**Objective 2: Identify Key System Functionality**

This is the second objective in the Inception phase, and you should work on it as you identify your use cases. It is important to decide which use cases are the most essential or architecturally significant to ensure that you spend more time up front on the most critical use cases.

The project manager and architect should work hand-in-hand on this activity, involving other stakeholders (such as the customer) as necessary, and using several criteria to determine which use cases are critical.

**A. The functionality is the core of the application, or it exercises key interfaces of the system,** and will hence have a major impact on the architecture. Typically an architect identifies these use cases by analyzing redundancy management strategies, resource contention risks, performance risks, data security strategies, and so on. For example, in a Point-Of-Sale system, Check Out and Pay would be a key use case because it validates the
interface to a credit-card validation system—it is also critical from a performance and load perspective.

**B. The functionality must be delivered.** The functionality captures the essence of the system, and delivering the application without it would be fruitless. Typically the domain and subject-matter experts know what this functionality is from the user perspective (primary behaviors, peak data transaction, critical control transactions, and so on). For example, you cannot deliver an order-entry system if you cannot enter an order.

**C. The functionality covers an area of the architecture that is not covered by any other critical use case.** To ensure that you address all major technical risks, you need to have a good enough understanding of each area of the system. Even if a certain area of the architecture does not seem to be of high risk, it may conceal unexpected technical difficulties that can be exposed by designing, implementing, and testing some of the functionality within that part.

Items A and C will be of greater concern to the architect; project managers will focus mainly on items A and B.

For a system with 20 use cases, typically 3 to 4 of them are critical.[1] During Inception, it is important to understand all the critical use cases you identify and provide a fairly detailed description of them. However, you may postpone describing some of the alternative flows for these critical use cases until later in the project, as long as they do not have a major impact on the architecture.

[1] Note that for some systems, one or two use cases may constitute the core of the application, with a larger number of "supporting" use cases enabling execution of the core use cases. In this situation, fewer than 20 to 30 percent of use cases are architecturally significant, and we would typically implement several scenarios for each core use case.

The critical use cases are listed in the Software Architecture Document (SAD; see Chapter 16).

For each of our three example projects, you do the following:

- **Project Ganymede, a small green-field project:** The architect and the project manager is the same person. The architect/project manager suggests 4 of the 15 identified use cases as being critical. After discussion with the customer, a fifth use case is added. The architect/project manager gets the entire team together and spends an hour explaining why these are the most critical use cases. The team agrees, with potential changes made, and the architect/project manager documents the critical use cases in the SAD.

- **Project Mars, a large green-field project:** The architect proposes a list of 8 of the 40 use cases as being architecturally significant, strictly from a technical risk mitigation standpoint. The project manager suggests a set of 9 use cases that are critical to the stakeholders. The project manager would like stakeholder buy-in on the functionality of these as soon as possible. Five of the use cases overlap. After a few days of discussion between the architect and project manager, the project manager drops 2 use cases from the list (the project can delay getting feedback on those use cases from the users). The architect sees a way to mitigate some of the risks in 8 of the use cases through some of the use cases the project manager added. They end up with a joint list of 9 use cases, which the architect
documents in the SAD.

- Project Jupiter, a second generation of a large project:
A lot of time will be spent on improving existing use cases, which means that fewer use cases will be critical than for green-field development. The architect identifies one of the existing use cases as critical because it involves using some unexplored new technology. The architect also identifies 2 of the 9 new use cases as being architecturally significant. The project manager identifies one additional use case as critical from the user perspective. The architect documents the 4 critical use cases in the SAD.

Objective 3: Determine at Least One Possible Solution

Since the overall goal of Inception is to determine whether it makes sense to continue with the project, you need to make sure that there is at least one potential architecture that will allow you to build the system with a sensible amount of risk and at reasonable cost. As an example, you may consider three options for a client/server architecture (see Figure 6.3). By analyzing desired functionality (in the first version, as well as future versions of the application), compatibility with other applications, and requirements on operations and maintenance, you may conclude which of these three options are viable. As you explore options, ask the following questions:

- What other, similar systems have been built, and what technology and architecture did you use? What was your cost?

- In particular, for an evolution of an existing system, is the current architecture still satisfactory, or does it need to evolve?

- What technologies would you have to use within the system? Do you need to acquire any new technologies? What are the costs and risks associated with that?

- What software components are needed within the system (database, middleware, and so on)? Can they be purchased? Can they be reused from another in-house project? What are the estimated costs? The associated risks?

Figure 6.3. Three Options for a Client/Server Architecture. During Inception, identify the type of architecture you intend to have and make implementations of necessary elements to the architecture to understand what risks you are facing. Here you see three options for a client/server architecture, each with vastly different requirements for tooling, competency, and complexity, and with different ability to address existing and future requirements on functionality, operation, and maintenance cost.
In some cases, you may need to acquire or implement some key elements of the architecture, or different suggested architectures, to better understand the risks you are facing and the options you have. For applications where stakeholders might find difficulty envisioning the end product, you should also spend time on implementing some functional prototypes, sufficiently rich to verify that the Vision makes sense.

At the end of Inception, you should have a rough idea of what risks you are facing, especially in the areas of acquisition of technology and reusable assets, such as architectural framework, packaged software, and so on. During Elaboration, you may come up with a better architecture, and that is fine. It is during Elaboration that you will address the vast majority of the architecture- and technology-related risks you identified during Inception.

For each of our three example projects, you do the following:

- **Project Ganymede**, a small green-field project: The team builds a functional prototype of the use case that is considered the most critical. The functional prototype identifies some key architectural components, including one that you need to purchase. The team builds bits and pieces of functionality to understand how some new technology can be used, allowing you to better understand what can be delivered.

- **Project Mars**, a large green-field project: The team builds a conceptual prototype in the first iteration. This helps the interaction between the team and the customer, so the team can do a better job documenting what the customer wants. The second iteration focuses primarily on understanding what technology to use and associated risks. Some of the key building blocks are outlined, and fragmented implementations of a couple of the critical use cases are done to better understand what technology choices to make.

- **Project Jupiter**, a second generation of a large project: The team makes mock-ups
Objective 4: Understand the Costs, Schedule, and Risks Associated with the Project

Understanding what to build is key, but determining how to build it and at what cost is also crucial. To determine whether you should continue with a project, you need to understand roughly how much the project will cost. Most of the costs are related to what resources you will need and how long it will take to complete the project. Combine all of this knowledge with an understanding of the required functionality and its value to the users, and you can build a Business Case for the project. The Business Case documents the economic value of the product, expressing it in quantitative terms such as, for example, return on investment (ROI). The Business Case is the instrument you use to obtain adequate project funding. It also outlines the major unmitigated risks and therefore the level of uncertainty that remains within the project.

In many organizations, especially internal IT departments, the budget has already been set before the project gets to the IT department. In this case, you determine what can be delivered within the budget and the schedule.

For organizations developing software using a low-ceremony approach, the business case may take the form of a short memo or e-mail message, while high-ceremony projects require quite extensive business cases.

For each of our three example projects, you do the following:

- Project Ganymede, a small green-field project: The project manager/architect writes up a two-page memo, which functions as a business case, to the project sponsor. This provides the sponsor with sufficient information to understand the value of the expected investments.

- Project Mars, a large green-field project: The project manager produces an 8-page Business Case and a 12-page Software Development Plan (SDP), referencing project plans, risks lists, and other key management artifacts. The project manager arranges a half-day review meeting with key stakeholders to walk through the Business Case, risk list, Vision, and Software Development Plan (see the section Project Review: Lifecycle Objective Milestone that follows).

- Project Jupiter, a second generation of a large project: Compared to the first
iteration, there is typically less business risk involved in developing the second generation of an application. Roughly the same process is followed as for the first project but with less rigor. The project manager writes a four-page Business Case and produces a Software Development Plan, referencing project plans, risk lists, and other key management artifacts. The project manager arranges a two-hour review meeting with key stakeholders to walk through the Business Case, risk list, Vision, and Software Development Plan (see the section Project Review: Lifecycle Objective Milestone that follows).

Objective 5: Decide What Process to Follow and What Tools to Use

It is important that your team shares a common view of how it will develop software, that is, which process it will follow. You should make sure that you streamline the process to minimize unnecessary overhead and be sure that the process addresses the specific needs of your project. Small projects can make decisions on exactly what process to follow as they go along, but bigger projects may need to spend more time up front considering their process of choice.

The idea is to come up with a process and tool environment you think works in your first iteration. You deploy the process and tools in the second iteration, and get immediate feedback on what works and what does not. Based on the feedback, you update your process and tool environment, roll it out in the next iteration, and keep on iterating until you are satisfied with your environment.

Many organizations go with a "gut feeling" when deciding what process to adopt. Often, they wind up applying Band-Aids to problems without addressing the source of the wounds. A better approach is to assess your organization to understand where you are now, define where you want to be, and then decide how to get there, incrementally. In Chapter 10, we describe how you can customize the RUP product to fit the specific needs of your projects and how you produce a development case, a customized guide for selecting which parts of the RUP product to use and how to use them.

Once you have decided on a process, you can choose what tools to use. In some cases, the tool environment may already be decided, through a corporate standard, for example. If not, then you need to choose which Integrated Development Environment (IDE), requirements management tool, visual modeling tool, Configuration and Change Management tool, and so on to use. As we mentioned before, it is important that the tools do a good job of automating the process you choose; this may require that you customize tools and templates, set up project directories, and so on.

You also need to implement the process and tools in your projects, which we describe in Chapter 11.

For each of our three example projects, you do the following:

- Project Ganymede, a small green-field project: The team gets together and spends an hour agreeing on how it should work at large. After the meeting, the project manager/architect produces a RUP configuration that corresponds to what they agreed on and writes a one-page development case for the Inception phase, outlining which artifacts should be produced, what templates to use, and how to document the information. They
decide to wait until the beginning of Elaboration to detail how to work in Elaboration. The project manager/architect, being more experienced, functions as a mentor for the rest of the team, helping them with the adoption of the process and tools.

- Project Mars, a large green-field project: The project manager and architect spend time with a mentor, who guides them in what process to use. Once the mentor understands the needs of the project, the mentor spends a few days producing a RUP configuration and writing a development case for the project. The development case covers the entire project (even though more detail is provided for the Inception phase). Once done, the mentor uses the development case to influence what is taught in the training delivered to the team.

- Project Jupiter, a second generation of a large project: Most of the team members were involved when developing the first version of the application. They were happy with the process and tool environment, but had suggested a few improvements, which were documented at the end of the previous project. These suggestions for improvements are now implemented and rolled out to the team.

**Project Review: Lifecycle Objective Milestone**

At the end of the Inception phase is the first major project milestone, called **Lifecycle Objective Milestone**. At this point, you examine the lifecycle objectives of the project. The project should be aborted or reconsidered if it fails to reach this milestone. If your project is doomed to fail, it is better to realize this early than late, and the iterative approach combined with this milestone may force such an early epiphany.

The Lifecycle Objective Milestone review includes the following evaluation criteria:

- Stakeholder concurrence on scope definition and an initial cost/schedule estimate (which will be refined in later phases)

- Agreement that the right set of requirements has been captured and that there is a shared understanding of these requirements

- Agreement that the cost/schedule estimate, priorities, risks, and development process are appropriate

- Agreement that initial risks have been identified and a mitigation strategy exists for each

The project may be aborted or reconsidered if it fails to reach this milestone.

**Conclusion**
Inception is the first of the four lifecycle phases in the RUP. It is really all about understanding the project and getting enough information to confirm that you should proceed—or perhaps convince you that you shouldn’t. This translates into five basic objectives:

- Understand what to build.
- Identify key system functionality.
- Determine at least one possible solution.
- Understand the costs, schedule, and risks associated with the project.
- Decide what process to follow and what tools to use.

By focusing on these objectives, you can avoid getting lost in the rich set of activities and guidelines that the RUP provides. Use these guidelines to achieve the objectives and to avoid common pitfalls. Apply the guidance we provide in Chapter 3 to guide you in how much ceremony you want to apply to your project. Don’t try to produce all artifacts in the RUP product; focus on those that will help you reach your objectives.

Chapter 7. The Elaboration Phase

This chapter provides a general introduction to Elaboration, the second phase of the RUP lifecycle. This is the phase in which the differences between the waterfall and iterative approaches are most apparent: In particular, there is a radical difference in the types of activities performed in each of the approaches. The major advantages of the iterative approach will become clear: It addresses major risks, builds an early skeleton architecture of the system, and refines and evolves the project plans that were produced in Inception. These plans will continue to be revised throughout the project. In short, iterative development allows your project to adapt to the discovery of new or unknown issues.

Rather than describe each possible activity you could undertake in the Elaboration phase, we will focus on what you want to achieve—that is, the objectives of the Elaboration phase—and then provide guidance on how to achieve it. This will help you to stay focused on the most essential activities in an actual project, making it less likely that the project will derail or become mired in “analysis-paralysis,” that is, nonessential activities that prohibit real progress. Or worse, you could focus on developing the wrong artifact or useless artifacts, just because they are described in the RUP.

While you read this chapter, remember that you need to decide how formally you want to capture artifacts; this could be done in your head, on a whiteboard, or in a model or document. How formally do you want to review these artifacts? We will try to point out obvious shortcuts you can make if you work in small projects or low-ceremony projects, but in the end, nothing can replace common sense, and you will need to make a judgment call as to what fits your project best. This is something that you have captured in your Software Development Plan in Inception.
Objectives of the Elaboration Phase

As Figure 7.1 shows, Elaboration is the second of the four phases in the RUP approach. The goal of the Elaboration phase is to define and baseline the architecture of the system in order to provide a stable basis for the bulk of the design and implementation effort in the Construction phase. The architecture evolves out of a consideration of the most significant requirements (those that have a great impact on the architecture of the system) and an assessment of risks.

Figure 7.1. The Elaboration Phase. Elaboration is the second phase of the RUP lifecycle; it has a well-defined set of objectives and is concluded by the Lifecycle Architecture Milestone. Use these objectives to help you decide which activities to carry out and which artifacts to produce.

This general goal translates into four major objectives, each addressing a major area of risk. You address risks associated with requirements (are you building the right application?) and risks associated with architecture (are you building the right solution?). You also address risks associated with costs and schedule (are you really on track?), and finally you need to address risks related to the process and tool environment (do you have the right process and the right tools to do the job?). Addressing these risks ensures that you can move into the Construction phase with a minimum of risk and issues.

1. **Get a more detailed understanding of the requirements.** During Elaboration, you want to have a good understanding of a vast majority of the requirements, since most of them are only briefly described in Inception. This will allow you to create a more detailed plan. You also want to get buy-in from key stakeholders to ensure that these are the correct requirements. And finally, you want to gain an in-depth understanding of the most critical requirements to validate that the architecture has covered all the bases, something that can be achieved only by doing a partial implementation of these requirements.

2. **Design, implement, validate, and baseline the architecture.** You want to design, implement, and test a skeleton structure of the system. The functionality at the application level will not be complete, but as most interfaces between the building blocks are implemented during Elaboration, you can compile and test your architecture. This is referred to as "executable architecture" (see the section Objective 2: Design, Implement, Validate, and Baseline the Architecture that follows), to the extent that you can (and should) conduct some initial load and performance tests on the architecture. You make critical design decisions, including the selection of technologies, main components, and their interfaces; you assess the buy-versus-build options; and you design and implement architectural mechanisms and patterns.

3. **Mitigate essential risks, and produce more accurate schedule and cost estimates.**
During Elaboration, you address major risks. Most will be addressed as a result of detailing the requirements and designing, implementing, and testing the architecture. You also refine and detail the coarse project plan for the project (see Chapter 12 for more details on planning).

4. **Refine the development case and put the development environment in place.** You refine the process you defined for Inception to reflect lessons learned. You also continue to implement the software development tools you need for our project.

**Elaboration and Iterations**

In the Elaboration phase, many risks are mitigated by producing an executable architecture, that is, a subset of the most essential aspects of the system that allow you to very concretely demonstrate key capabilities and therefore unambiguously assert that the risks are eliminated. If you've already built a system with the same technology that you're using in your project, then you can often achieve this objective in a single iteration because there is a limited amount of risk to address. You can reuse solutions from the past and thus make rapid progress.

But if you're inexperienced in the application domain, if the system is very complex, or if you're using new technology, then you may need two or three iterations to get the architecture right and to mitigate key risks. Other factors that will lead you to require multiple iterations include doing distributed development, having many stakeholders or complex contractual agreements, or needing to comply with safety regulations or other external standards.

For each of our three example projects, you would typically have the following iterations pattern:

- **Project Ganymede, a small green-field project:** Because the application is small, you can typically get the architecture right in a short time. You probably need only one iteration, but if you have a lot of new technology and are building an unprecedented application, you may need two iterations.

- **Project Mars, a large green-field project:** Because the application is more complex and you have never built this type of system before, you need some time to understand and mitigate technical risks and get the architecture right. You probably need two, or maybe even three, iterations in Elaboration.

- **Project Jupiter, a second generation of a large project:** You are primarily adding features and making bug fixes without making major changes to the architecture. You will use some new technology and develop some new subsystems, but one iteration should be sufficient. If you are not making anything but minor changes to the architecture, you may not have any iteration at all.

Assuming that our large green-field project would have two iterations, the general aim is to
implement the use cases that are most essential to customers, as well as those associated with the most technical risk in the first iteration. Especially in the first iteration, and to some extent the second, start with only partial implementation of use cases (that is, implement only some of the scenarios within the use case) to quickly drive out as much risk as possible and to get a reasonable implementation before detailing the use cases. Each iteration could look as follows.

**First Iteration in Elaboration**

- Design, implement, and test a small number of critical scenarios to identify what type of architecture and architectural mechanism you need. Do this as early as possible to mitigate the most crucial risks.

- Identify, implement, and test a small, initial set of architectural mechanisms.

- Do a preliminary logical database design.

- Detail the flow of events of roughly half of the use cases you intend to detail in Elaboration, in order of decreasing priority.

- Test enough to validate that your architectural risks are mitigated. Do you, for example, have the right level of performance?

**Second Iteration in Elaboration**

- Fix whatever was not right from the previous iteration.

- Design, implement, and test the remaining architecturally significant scenarios. The focus in this iteration should be on ensuring architectural coverage (see the section Ensure Architectural Coverage later in this chapter for more information).

- Outline and implement concurrency, processes, threads, and physical distribution, as necessary, to address technically high-risk issues. Focus on testing performance, load, and interfaces between subsystems and external interfaces. (Note that planning what is done when is based on mitigating top risks early in the project. For some systems, resolving concurrency, processes, threads, and so on may entail very high risk. If that is the case, then you should already have started addressing these issues in the first iteration in Elaboration.)

- Identify, implement, and test remaining architectural mechanisms.

- Design and implement a preliminary version of the database.

- Detail the second half of the use cases that need to be specified in Elaboration.

- Test, validate, and refine your architecture to the point where it can be your baseline. A baseline means that you use the architecture as a stable reference point. You can still make changes to it, but you want to avoid major rework except to resolve critical problems. If you cannot reach a sufficiently stable state on your architecture, you should add another iteration to Elaboration. This is likely to delay the project, but it will cost you even more to continue building on quicksand, that is, to invest further in an architecture that keeps going through major changes.
Objective 1: Get a More Detailed Understanding of the Requirements

By the end of Inception, you should have produced a good **Vision** as well as a **detailed description of the 20 percent or so most essential use cases**, at least of the architecturally significant portions of these use cases. You also have a brief description (maybe two to three paragraphs) of the remaining use cases.

By the end of Elaboration, you will want to complete the description of a majority of use cases. Some use cases may be so simple, or so similar to other use cases but operating on other data, that you can comfortably postpone them until Construction, or even never formally describe them. Detailing them will not address any major risks. You should also produce a user-interface prototype for major use cases, if necessary, to make stakeholders understand what functionality the use case provides. Walk through and test each use case with a user, using the user-interface prototype to clarify what the user experience will be and what information is displayed and entered.

As you detail use cases, you are likely to find additional use cases, which then are added and prioritized.

You should also continuously update the glossary. In some cases, you may want to express graphically how different glossary items relate to each other. You do this by expressing the most important glossary items as "domain objects" in a small domain model (see Workflow Detail: Develop a Domain Model, in the RUP, for more information).

As we described in the section Detail Key Actors and Use Cases in Chapter 6, you typically want to "time-box" the activities dealing with use cases to avoid getting bogged down in too much detail. Also note that especially for small projects, you often have the same person take on the role of analyst and developer, meaning that the same person who describes a use case will also implement it. If this is the case, you may want to spend less time on documenting the detailed requirements and come back to them as you implement and validate the use case.

At the end of Elaboration, you will have detailed a vast majority of the requirements (probably about 80 percent). As more and more use cases are implemented during Construction, you will refine each use case as necessary. You may find additional use cases during Construction, but that should be more of an exception than the rule.

For each of our three example projects, you do the following:

- **Project Ganymede**, a small green-field project: The team finds another use case and another actor. They spend another two hours per use case describing 9 out of the 12 use cases not yet detailed (see the section Detail Actors and Use Cases, in Chapter 15).

- **Project Mars**, a large green-field project: In the first iteration, the team refined the Vision and found 3 more use cases, making the total number of use cases 43. They described in detail another 12 use cases, adding to the 9 they already created (see the section Detail Actors and Use Cases, in Chapter 15). In the second iteration, they found 1 more use case,
but decided to make 2 of the current use cases "out of scope." They also described in detail another 13 use cases, adding to the 21 they already had done.

- Project Jupiter, a second generation of a large project: They updated the Vision, added a use case, and described in detail most of the use cases not yet detailed (see the section Detail Actors and Use Cases, in Chapter 15). They analyzed in detail fixes to major defects that must be corrected and analyzed the impact on the architecture.

Objective 2: Design, Implement, Validate, and Baseline the Architecture

Software architecture and the related artifacts and activities are also described in Chapter 16. For now, let us simplify architecture to a few key design choices that must be made:

- The most important building blocks of the system, and their interfaces, as well as the decision to build, buy, or reuse some of these building blocks

- A description of how these building blocks will interact at runtime to implement the most important scenarios you have identified

- An implementation and testing of a prototype of this architecture to validate that it does work, that the major technical risks are resolved, and that it has the proper quality attributes: performance, scalability, and cost

In the RUP, an architecture is not limited to a paper drawing or a set of blueprints. To validate an architecture, you need more than a review of a paper representation; you need an executable architecture that can be tested to verify that it will address your needs and that it constitutes the proper basis on which to implement the rest of the system.

An executable architecture is a partial implementation of the system, built to demonstrate that the architectural design will be able to support the key functionality and, more important, to exhibit the right properties in terms of performance, throughput, capacity, reliability, scalability, and other "ilities." Establishing an executable architecture allows the complete functional capability of the system to be built on a solid foundation during the Construction phase, without fear of breakage. The executable architecture is built as an evolutionary prototype, with the intention of retaining validated capabilities and those with a high probability of satisfying system requirements when the architecture is mature, thus making them part of the deliverable system.

Note that in the layered architecture shown in Figure 7.2, the elements in the lower layers either already exist or will be built during Elaboration; the application layers will be fleshed out with production code during Construction, but will be only partially completed during Elaboration (perhaps only to the extent of constructing the subsystems shells). Nevertheless, you need to do performance- and load-testing of the critical scenarios by the end of Elaboration, and you should write "stubs" (if not actual application code) to enable this end-to-end testing.

Figure 7.2. The Architecture Provides a Skeleton Structure of Your Application. This figure shows one possible representation of the static structure of your system. The architecture consists of the most essential
**building blocks and their interfaces, and specifies common solutions to frequently encountered problems. It provides a skeleton structure of the application, leaving you to fill in the blanks within a stable and well-defined base.**

![Diagram showing structure of an application](image)

At the end of the Elaboration phase, you baseline your architecture, which means that you make your architecture a stable reference for building the rest of the system. From this point on, you should modify the architecture with care, and only if you have a good reason. This baseline provides some stability for the development team. Note that the larger your team and the more technically complex the project, the more important it is to baseline the architecture. The smaller the team and the less complex your architecture is, the more liberty you can take in modifying the architecture.

See [Chapter 16](#) for more information on architecture.

### Architecture: Defining Subsystems, Key Components, and Their Interfaces

At the end of Inception, you produced or at least identified one potential architecture that would allow you to build the system with a reasonable amount of risk and at a reasonable cost. In some cases, you also implemented key elements of the architecture, which we described in the section Objective 3: Determine At Least One Possible Solution, in [Chapter 6](#).

At the end of Inception, you had a rough idea of what risks you were facing, especially in the areas of acquisition of technology and reusable assets, such as architectural framework, packaged software, and so on. You left the majority of questions unanswered; other answers were preliminary and left to be finalized during Elaboration.

For all these reasons, early in the Elaboration phase you should have a fairly good understanding of what kind of system you are building. Rather than inventing an architecture, you should first envisage using an existing architectural framework to advance the architecture. Maybe there is a
commercial framework available (for example, IBM's IAAA for insurance applications), or perhaps you have built this type of system before and can harvest the architecture from previous work. If not, then you need to identify the major building blocks, that is, the subsystems and major components. Potential sources of input are the major abstractions captured in the domain object model or glossary (see Chapter 6). For example, an online transaction system typically requires a component or subsystem that handles each major concept: Shopping Cart, Customer, and Price Promotions. For each identified subsystem or component, describe the key capabilities they need to offer, namely, their interfaces toward the rest of the system.

[1] A subsystem corresponds to a component or a collection of components.

In parallel with identifying key components and subsystems, you need to survey available assets inside and outside the company. Can you acquire a set of components implementing the concept of Shopping Cart? Do those components satisfy your needs? What are the legal and financial implications? Will the components be maintained as technology and user requirements evolve? Do we have access to the source code to make necessary changes? Is the code properly documented with supporting guidelines about the components' design and how to use and test them?

Use Architecturally Significant Use Cases to Drive the Architecture

During Inception, you should have identified some use cases, perhaps 20 to 30 percent, as being critical to the system (see the section Objective 2: Identify Key System Functionality, in Chapter 6, for more information). They are also likely to be significant in driving the architecture.

[2] Note that for some systems, one or two use cases may constitute the core of the application, with a larger number of “supporting use cases” enabling execution of the core use cases. In this situation, fewer than 20 to 30 percent of use cases are architecturally significant, and you would typically implement several scenarios for each core use case.

You also need to identify certain elements in the requirements—possibly nonfunctional requirements—that are difficult, unknown, or at risk, and find use cases (or fragments of use cases) that would illustrate the difficult points and whose implementation would force confrontation and resolution of the risk. These are the technical challenges often relegated to the infrastructure part of the architecture. For example, if there is a very demanding response time requirement, or load requirements, identify one use case (or just one flow of events in one use case) that would illustrate this requirement, together with the expected performance requirement. Other examples would be an error recovery strategy or system startup.

Finally, you should identify some use cases that, although not critical nor technically challenging, address some parts of the system not yet covered, so that you will develop a good grasp over the whole architecture of the system by the end of Elaboration. For example, make sure that all your major "business entities" are exercised by at least one of your architecturally significant use cases.

You must ensure that the architecture will allow you to deliver all the architecturally significant use cases by designing, implementing, and testing as many of these use cases as necessary to mitigate the risks associated with them (see Figure 7.3). At the same time, you do not want to implement more capabilities than necessary to mitigate the risks, since that would take time from other activities related to risk mitigation, which is one of your primary objectives in Elaboration.

**Figure 7.3. Architecturally Significant Use Cases Drive the Architecture.**

*For most systems, you can drive out a majority of technical risks and drive the implementation of the architecture by choosing the right 20 to 30*
percent of use cases, and designing, implementing, and testing one or two scenarios for each use case. To implement a given use case, you need to identify which software elements are required to provide the functionality of that use case.

This typically means that, in Elaboration, you should focus on only one or two scenarios or flows of events in a use case: Typically, you would choose the basic flow of events, or "Happy Day" scenario. If necessary, you may also need to implement some scenario(s) involving unexpected events. For example, you might implement one scenario to eliminate risk associated with exception handling, but there would be no point in implementing 10 scenarios to mitigate this same risk.

Design Critical Use Cases

The design representation of a use case is called use-case realization (see Figure 7.4). It describes how a particular use case is realized within the design model, in terms of collaborating objects. You can divide this work into an analysis section and a design section. The following provides an overview of the five most essentials steps when producing a use-case realization. It should be noted that these steps are helpful for all developers, but you need to determine from project to project how formally you want to document the results of each step, such as on a whiteboard or in a visual modeling tool. See the section Design Use-Case Realizations and Components, in Chapter 17, for a more detailed explanation of these steps.

1. Make a preliminary outline of the analysis objects involved in the collaboration. The RUP product provides some excellent guidance to assist those developers inexperienced in object-oriented development in identifying good analysis classes.
2. **Distribute behavior to analysis classes.** That is, specify the overall responsibility of each analysis class so you understand how these classes together can provide the functionality of the use case.

3. **Detail analysis classes.** This helps you understand the responsibility of each analysis class. Review the analysis model to ensure that no two classes provide overlapping functionality and that the relationships between various classes make sense.

4. **Design use cases.** In other words, specify in exactly what order, and how, each design class will communicate with other design classes to provide the architecturally significant parts of the use-case functionality. This way of partitioning the functionality of a use case into a set of design elements that communicate with each other can be used for object-oriented or nonobject-oriented systems.

5. **Refine analysis classes into design classes.** In many cases, several design classes implement one analysis class. Detail each design class by specifying operations and attributes, review the design model to ensure that you have not duplicated functionality across several classes, and see that all relationships between classes make sense.

**Figure 7.4. An Example Sequence Diagram.** A use-case realization shows how your design elements are collaborating to provide the functionality of the architecturally significant parts of the use case. One way to show this collaboration is through a sequence diagram.

Simple use cases with limited sequencing, especially if implemented via a powerful programming language (such as Visual Basic or another fourth-generation language), typically do not require all these steps, especially step 4.
Consolidate and Package Identified Classes

The next step is to group the identified classes into appropriate subsystems. The architecture team will already have identified some of the subsystems (see the earlier section Architecture: Defining Subsystems, Key Components, and Their Interfaces). Some guidelines for packaging classes follow:

- **Localize the impact of future changes by grouping classes that are likely to change together into the same subsystem.** (See Figure 7.5.) For example, if you expect the interface for a certain actor to change radically, place all the classes implementing the interface for that actor in the same subsystem.

  **Figure 7.5. Packaging Should Localize Impact of Change. You can localize the impact of change by placing all classes that are dependent on a certain database, or the interface for an actor, in the same subsystem.**

- **Enforce certain rules of visibility.** For example, enforce rules that define boundaries between the multiple tiers in a client/server application. You do not want to package classes from different layers into the same subsystem.

- **Consider packaging classes according to how you will configure the future application/product.** This means that you can assemble various configurations of the final application by choosing to include or exclude various subsystems.

For more considerations regarding how to package classes, see Guidelines: Design Package in the RUP product.

Ensure Architectural Coverage

One important objective in building the executable architecture is to ensure that it includes use cases touching on all major areas of the system. This ensures that a seemingly straightforward area of the system does not hide unexpected problems, an issue of particular importance when building unprecedented systems. Once you have consolidated the packaging and detailed the use-case realizations, you need to confirm that all areas of your system are covered. If, toward the end of Elaboration you discover "untouched" areas of the system, you should identify additional scenarios to implement in order to ensure architectural coverage (see Figure 7.6). This is part of your risk mitigation strategy to minimize unexpected issues later. A good coverage will also ensure
that your estimates are valid.

**Figure 7.6. Architectural Coverage. Use case E may not be considered architecturally significant, but it is added to the list of use cases to design, implement, and test to ensure that you have architectural coverage for otherwise untouched parts of the architecture.**

Architectural coverage is typically more a concern for larger projects than for smaller projects and can often be disregarded by small projects.

**Design the Database**

Many systems have a database, and you need to understand how persistent data is to be stored and retrieved. You can find comprehensive guidance within the Rational Unified Process in the area of database design (see the Database Design activity and the Data Model guidelines in the RUP product). You can also find useful information in [Ambler 2000](#).

**Outline Concurrency, Processes, Threads, and Physical Distribution**

Next, you need to describe the run-time architecture in terms of concurrency, processes, threads, interprocess communication, and so on. For distributed systems, you need to describe the distribution by outlining the physical nodes. This description includes defining the network configuration and allocating processes to nodes. We discuss this issue in more detail in [Chapter 16](#).

**Identify Architectural Mechanisms**
Architectural mechanisms represent common concrete solutions to frequently encountered problems (see Figure 7.7). They are architectural patterns, providing standard solutions to problems such as garbage collection, persistent storage, list management, "shopping cart" implementation, or communication with external systems through a certain protocol.

**Figure 7.7. Architectural Mechanisms.** *Architectural mechanisms provide solutions to common problems. You may have one or several mechanisms for persistency, communication, parsing, and authentication; each one may be used many times in the system.*

By designing, implementing, testing, and documenting architectural mechanisms, you can solve the most common and difficult problems once, and then all team members can take advantage of these ready-made solutions whenever they need them. This approach allows developers to be more productive, and it greatly speeds up the work in the Construction phase, when typically more people join the project.

We also discuss architectural mechanisms in Chapter 16.

**Implement Critical Scenarios**

Each design class provides a specification for code. In most cases, implementing the class is done iteratively as the class is designed. You design a little, implement what you design, detect deficiencies, and then improve the design. As you implement components, you need to unit-test the component to ensure that it performs according to specifications and that you have not introduced memory leaks or performance bottlenecks (see the section Developer Testing, in Chapter 17, for more details).

When both designing and implementing a class, you need to consider how to test the system. You might also need to design and implement test classes representing test drivers and interfaces to automated test tools.

**Integrate Components**
When doing iterative development, it becomes increasingly complex to plan builds and integration testing. In parallel with identifying analysis classes, you need to determine in what order you will integrate what components, and as you do class design, you need to verify that you design and implement the functionality necessary to integrate and compile your evolving system for testing.

Integration is a continuous activity. If your iterations are four weeks long, for example, you should typically produce a build daily or at least twice weekly. As the size of your system and your team grow, you may have to increase the interval between builds (as well as the iteration length). Note that the level of support you have for configuration management, including automated build management, highly impacts your ability to plan and frequently create builds.

**Test Critical Scenarios**

Testing is an extremely important aspect of Elaboration. The best way to verify that you have mitigated risk is to test the executable architecture. Among other things, you want to verify that

- **Critical scenarios have been properly implemented and offer the expected functionality.**

- **The architecture provides sufficient performance.** Typically there are a couple of scenarios critical to performance, and these need to be performance-tested. For an Online Transaction System, you might need to verify that the use case Check Out performs sufficiently. If it does not, then you may have to rework the architecture.

- **The architecture can support necessary load.** Usually, there is a small set of scenarios that are critical to load, and these need to be load-tested. For an Online Transaction System, you might need to verify that the use cases Browse Catalog, Put Item in Cart, and Check Out can carry a load of 1,000 simultaneous users. If it can't, then you might have to revisit various architectural decisions.

- **Interfaces with external systems work as expected.** Does the API work as expected? What about performance and synchronization issues?

- **Any other requirements in the supplementary specification (nonfunctional requirements) that are not captured above are tested.** Failover, reconfiguration, and recovery scenarios might fit into this category. The supplementary specification is an important source of requirements that need to be tested from an architectural viewpoint and might require you to construct special scenarios for effective testing.

Some of this testing can be automated with test tools, allowing you to see whether you lack the proper test tools or lack competency with the tools you have. In this way, the iterative approach forces you to “test” your test capabilities early on so that you can fix any issues before it is too late.

For more information on testing, see Chapter 18.

**What Is Left to Do?**

The list of activities we've just covered is quite comprehensive, so what is left to do? Well, keep in mind that although you have completed many types of project activities, you have only designed, implemented, and tested 10 to 20 percent of the system. You partially implemented only 20 to 30 percent of the use cases, and then implemented only one or two Happy Day scenarios for each of
those use cases. You may also have implemented some architecturally significant alternative flows of events, among others, to test your exception mechanisms.

But overall, the majority of the coding effort for the project will deal with alternative or unexpected user interaction and exception handling. So, in Elaboration, you did a little of everything, but you still have roughly 80 percent of code left to do in the following lifecycle phases of Construction and Transition. On the positive side, however, the code you did implement and test represents the most difficult parts of the system, and it allows you to mitigate the vast majority of risks in your project.

For each of our three example projects, you do the following:

- **Project Ganymede, a small green-field project:**
  The team evolves the functional prototype built in Inception into a more complete executable architecture, allowing them to showcase some of the key functionality (when in the hands of developers, only certain very well-defined scenarios are supported) and, more important, confirm that the architecture supports the necessary performance, scalability, dependability, and so on. The architecture was tested and used as a baseline for further work.

- **Project Mars, a large green-field project:**
  The team evolved the conceptual prototype built in Inception into a more complete executable architecture, allowing them to showcase some of the key functionality (when in the hands of developers, only certain very well-defined scenarios are supported) and, more important, to confirm that the architecture supports the necessary performance, scalability, dependability, and so on. Since performance and scalability were a big issue, a fair amount of time was spent on doing performance- and load-testing of the architecture. This exposed a number of issues with the architecture. A lot of rework was done, which saved a great deal of time down the road. The revised architecture was used as a baseline for further work. The architect walked through the architecture with the entire team to ensure that everybody understood the architecture.

- **Project Jupiter, a second generation of a large project:**
  The team could rapidly go through Elaboration because they did not face major technical risks. Technical risks were mitigated by implementing and trying out some of the new technology. Partial implementations of a few key use cases were done to verify that the new functionality would not regress the architecture.

**Objective 3: Mitigate Essential Risks, and Produce Accurate Schedule and Cost Estimates**

During Elaboration you mitigate the vast majority of technical risks—risks associated with understanding and getting buy-in on user requirements and risks associated with getting the project environment up and running. We discuss risk management in more detail in Chapter 14.

**Plan the Project and Estimate Costs**
Toward the end of Elaboration, you have more accurate information allowing us to update our project plan and cost estimate.

- You have **detailed the requirements** so you understand what system you are building. You update the Vision accordingly.

- You have **implemented a skeleton structure (executable architecture)** of the system, which means that you have solved many of the most difficult problems; you are primarily left with filling in the holes within a large set of well-defined areas. (You should not underestimate the amount of work left, but at least you know what is left to do.)

- You have **mitigated the vast majority of risks**. This radically reduces the gap between lower- and upper-range estimates for schedule and cost.

- You understand how effectively you are working with the people, the tools, and the technology at hand because you have used all three to go through the full lifecycle at least once (once for each iteration in Elaboration).

See [Chapter 12](#) for more information on planning a project.

For each of our three example projects, you do the following:

- **Project Ganymede**, a small green-field project: The project manager/architect spends a couple of hours updating the estimates on cost and schedule and writes a memo with risks and how to mitigate them. The project manager/architect spends 30 minutes with the team explaining the information and sends the information in an e-mail message to the project sponsor.

- **Project Mars**, a large green-field project: The project manager updates the Business Case, the Software Development Plan, with attached project plans, risks lists, and other key management artifacts. The project manager arranges a half-day review meeting with key stakeholders to walk through the Business Case, risk list, Vision, and Software Development Plan (see the section [Project Review: Lifecycle Architecture Milestone](#), below).

- **Project Jupiter**, a second generation of a large project: The project manager updates the Business Case and the Software Development Plan with attached project plans, risks lists, and other key management artifacts. The project manager arranges for a two-hour review meeting with key stakeholders to walk through the Business Case, risk list, Vision, and Software Development Plan (see the section Project Review: Lifecycle Architecture Milestone, below).

---

**Objective 4: Refine the Development Case, and Put the Development Environment in Place**
During Inception, you defined what process to follow and documented your way of using the RUP approach in a development case. You also defined what tools to use and did necessary tool customizations. In Elaboration, you walked through the full lifecycle, doing some design, implementation, and testing of the architecture. You also put your code base under Configuration Management.

To support these activities, you complete the installation and rollout of the process and tools that you initiated, and as you walk through the lifecycle, you learn both what works well for your project and what does not work well. You understand how to improve the process and what tuning and further customizations are necessary for your tools. You update your development case accordingly and fine-tune your tool implementation.

For each of our three example projects, you do the following:

- Project Ganymede, a small green-field project: The team members get together and spend an hour discussing how they liked the process and tool environment used in Inception. After the meeting, the project manager/architect updates the development case to cover the Elaboration phase as well, outlining what artifacts should be produced, what templates to use, and how to document the information. Also in this phase, the project manager/architect functions as a mentor for the rest of the team, helping them with adopting the process and tools.

- Project Mars, a large green-field project: The mentor of the project talks with various team members to get some feedback on what worked well and what did not work well during Inception. Based on the feedback, the mentor updates the development case for the project. The mentor uses the development case to influence any training delivered during Elaboration.

- Project Jupiter, a second generation of a large project: The project manager talks with various team members to get some feedback on what worked well and what did not work well during Inception. Based on the feedback, the project manager updates the development case and walks through any updates with the team. Most team members are familiar with the process and tools, so no training is needed.

Project Review: Lifecycle Architecture Milestone

At the end of the Elaboration phase is the Lifecycle Architecture Milestone. At this point, you examine the detailed system objectives and scope, the choice of architecture, and the resolution of the major risks. If the project fails to reach this milestone, it might be aborted or at least seriously reconsidered, and it's better for this to happen early, rather than late. The iterative approach, in combination with this milestone, forces such a decision.
The Lifecycle Architecture Milestone review includes the following evaluation criteria:

- Are the product Vision and requirements stable?
- Is the architecture stable?
- Are the key approaches to be used in testing and evaluation proven?
- Have testing and evaluation of executable prototypes demonstrated that the major risk elements have been addressed and resolved?
- Are the iteration plans for Construction of sufficient detail and fidelity to allow the work to proceed?
- Are the iteration plans for the Construction phase supported by credible estimates?
- Do all stakeholders agree that the current Vision, as defined in the Vision Document, can be met if the current plan is executed to develop the complete system in the context of the current architecture?
- Are actual resource expenditures versus planned expenditures acceptable?

For large projects, this review may take the form of a one-day assessment, or it may be performed over several days. Smaller projects may do the assessment in a one-hour team meeting.

Conclusion

At the end of Elaboration, the second of the RUP approach's four phases, you can look back and see that you made considerable progress, compared to where you were at the end of Inception. Here are the major achievements:

- You moved from a high-level understanding of the most important requirements to a detailed understanding of roughly 80 percent of the requirements.
- You moved from a potential and probably conceptual architecture to a baselined, executable architecture. This means you designed, implemented, and validated the architecture—a skeleton structure of the system—then produced a baseline of it.
- You mitigated a majority of architecturally significant risks and produced more accurate schedule/cost estimates for the remaining lifecycle phases. You used the Lifecycle Architecture Milestone to decide whether you should move ahead with the project, cancel it, or radically change it.
- You refined the development case and put the development environment in place.
- You had a small team of your most skilled people tackle the most difficult activities; you laid
the groundwork for successful continuation of the project and for scaling it up with a minimum of financial, business, and technical risks.

Chapter 8. The Construction Phase

This chapter describes what Construction, the third phase of the Rational Unified Process lifecycle, is all about (see Figure 8.1). While you read this chapter, remember that you need to decide how formally you want to capture artifacts: This could be done in your head, on a whiteboard, or in a model or document. Nothing can replace common sense, and you will need to make a judgment call as to what fits your project the best.

**Figure 8.1. The Construction Phase.** *Construction is the third phase of the RUP lifecycle; it has a well-defined set of objectives and is concluded by the Lifecycle Architecture Milestone. Use these objectives to help you decide which activities to carry out and which artifacts to produce.*

Elaboration ended with the internal release of a baselined, executable architecture, which allowed you to address major technical risks, such as resource contention risks, performance risks, and data security risks, by implementing and validating actual code. During the Construction phase, you focus heavily on detailed design, implementation, and testing to flesh out a complete system.

So what is left to do? Actually, the vast majority of the work. Many use cases have not been implemented at all, and the ones that have are typically only partially implemented—just enough to validate some hypothesis or to mitigate a risk. Subsystems have been defined and interfaces implemented, but only a very small subset of the underlying code (handling the business logics, the alternative flows of events, and error handling) has been implemented. As you implement more and more functionality, you will fine-tune the requirements. So, among other things, there is a lot of requirements tuning and detailed design, implementation, and testing to be done in Construction (see Figure 8.2).

**Figure 8.2. Work Distribution over the RUP Phases.** *In Construction, the emphasis shifts toward requirements tuning and detailed design, implementation, and testing. You also have a strong focus on deployment toward the latter part of the phase. Configuration and Change Management are now used routinely on all aspects of the project.*
In fact, the Construction phase is typically the most time-consuming. As we will discuss in more detail in the section Staffing the Project, in Chapter 12, on average, 65 percent of overall effort and 50 percent of overall schedule time are spent in Construction (note that these figures can vary greatly from project to project).

Even though most major technical risks were addressed during Elaboration, new risks will continually pop up in Construction, which you need to address. In general, however, these risks should have a smaller impact on the overall architecture. If not, it indicates that you did not do a good job during Elaboration.

During Construction, you focus on developing high-quality code cost-effectively. You take advantage of architectural mechanisms to accelerate the production of code. Especially for larger projects, ensuring architectural integrity, parallel development, Configuration and Change Management, and automated testing become primary skills to ensure success. You leverage the baselined architecture that you produced in Elaboration to allow you to scale up the project and add additional people as required.

As you design, implement, and test more and more functionality, you will continuously renegotiate especially detailed requirements to determine what is the right functionality to deliver. You will increasingly focus on what is the right balance between quality, scope, time, and "feature polishing" (choosing the right balance between producing barely usable features and perfecting a solution that already is good enough—also referred to as "gold plating").

### Objectives of the Construction Phase

Construction is really about cost-efficient development of a complete product—an operational version of your system—that can be deployed in the user community. This translates into the following objectives:

1. **Minimize development costs and achieve some degree of parallelism** in the work of the development teams. Optimize resources and avoid unnecessary scrap and rework. Even
smaller projects generally have components that can be developed independently of one another, allowing for natural parallelism between developers or teams of developers (resources permitting).

2. **Iteratively develop a complete product that is ready to transition to its user community.** Develop the first operational version of the system (beta release) by describing the remaining use cases and other requirements, filling in the design details, completing the implementation, and testing the software. Determine whether the software, the sites, and the users are all ready for the application to be deployed.

## Construction and Its Iterations

The number of iterations required for Construction varies from project to project, but in most cases Construction has more iterations (usually two to four) than any other phase (see the section Determining the Number of Iterations, in Chapter 12).

So what goes into each of the iterations? Iteration planning is largely driven by the parts of use cases that should be implemented. You want to implement the use cases that are most essential to customers, as well as those associated with the most technical risk. Especially in the first iteration, and to some extent the second, start with only partial implementation of use cases (that is, implement only some of the scenarios within the use case) to drive out quickly as much risk as possible and to get a reasonable implementation before detailing the use cases. Once you have decided which use cases to implement, or partially implement, identify which components need to collaborate to provide the use-case functionality; these are the components that must be further designed, implemented, and tested within that iteration. This identification provides you with a better understanding of the time required to implement the use cases and whether, based on available resources, the scope of work needs to be changed for the given iteration.

Let's assume that you have 15 use cases and your project has three iterations in Construction. How do you proceed? Table 8.1 shows a possible plan, starting with what has been achieved coming into Construction (results at the end of Elaboration) and what is done in each of the three Construction iterations.

### Table 8.1. Progress Made Prior to and Through Construction Iterations.
*The requirements, components, and the testing of the system evolve in each iteration. By the end of the Construction phase, you have the first operational version (beta release) of the system.*

<table>
<thead>
<tr>
<th>Requirements</th>
<th>Components</th>
<th>Tests</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>End of Elaboration</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• 15 UCs identified</td>
<td>• 18 main components identified</td>
<td>• Initial performance- and load-test of architecture has been done, primarily driven by architecturally significant UCs</td>
</tr>
<tr>
<td>• 8 UCs described in detail, 4 with some depth, 3 just briefly</td>
<td>• 4 have 50% of the code implemented, including all interfaces</td>
<td>• Functionality of 4 architecturally significant UCs has been properly</td>
</tr>
<tr>
<td>• 10 have interfaces plus minimal logic implemented</td>
<td></td>
<td></td>
</tr>
<tr>
<td>End of the first iteration in Construction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td></td>
</tr>
<tr>
<td>- 12 UCs described in detail, 3 with some depth</td>
<td>- 18 main components identified (one was not needed due to elimination of a UC)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- 10 have been almost completely implemented</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- 8 have 50% of the code implemented, including all interfaces</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- 8 have interfaces plus minimal logic implemented (approximately 10–20% of final code)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Lower layers in a layered architecture have been completely implemented</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Implemented code has been unit-tested</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Performance- and load-test of the system are continued to ensure architecture has not been compromised</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Functional testing of UCs is done as they are completed</td>
<td></td>
</tr>
</tbody>
</table>

| End of second iteration in Construction | 
| --- | --- |
| - 1 of the 3 UCs not yet described is scoped out due to time constraints | - 18 main components identified (one was not needed due to elimination of a UC) |
| | - 10 have been almost completely implemented |
| | - 8 have 50% of the code implemented, including all interfaces |
| | - implemented code has been unit-tested |
| | - Performance- and load-test of the system is continued to ensure architecture has not been compromised |
| | - Functional testing of UCs is done as they are completed |
### Objective 1: Minimize Development Costs and Achieve Some Degree of Parallelism

Developing software in a cost-effective manner is the objective of all phases. If you have only started to think about it in Construction, then you will fail to achieve it. However, if you have done Elaboration correctly—that is, if you have created a baselined, robust executable architecture—you can develop a more cost-effective software than would otherwise be the case. This is possible since you can reuse architectural mechanisms; you can work in parallel by organizing around the architecture (see the following section, Organize Around Architecture), and you will run into fewer unexpected problems since you have mitigated many major technical risks. In this section, we will look at some of the things that especially larger teams should consider in Construction (and possibly earlier) to ensure successful software development.

#### Organize Around Architecture

One of the many benefits of a robust architecture is that it clearly divides the system responsibilities into well-defined subsystems. You have an architect or an architecture team worrying about the architecture and how it all ties together, and individuals can focus primarily on their assigned subsystem(s). Developers still need to understand the overall system, but they can focus primarily on a subset of a system, that is, one or several subsystems assigned to them. Organizing around the architecture prevents people from stepping on each other's feet.

Organizing around the architecture also helps with communication. Face-to-face communication is normally the most effective form of communication, but as projects grow bigger, you need to reduce the need for communication since the face-to-face method does not scale well. The upper half of Figure 8.3 shows how many possible communication paths there are among all team members (note that it grows geometrically with the size of the team). For a team of size N, the number of communication paths = N * (N−1) / 2. This means that a two-person team has 1 communication path, a three-person team has 3 communication paths, but a six-person team has 15 communication paths.
Figure 8.3. Organization Around Architecture Minimizes Communication Overload. The possible communication paths between team members grows geometrically with the size of the team. Organizing around the architecture radically reduces the number of communication paths within a team. Issues regarding subsystem interaction are resolved by the architecture team, which owns the interfaces between subsystems.

An increase in communication paths destroys project team efficiency, and you need to find better means of communication (other than everybody communicating with everybody). This can be achieved by having one team responsible for the architecture, and several small teams each responsible for one or several subsystems. Communication between these few teams is channeled through the architecture team to resolve issues around the overall solution and interfaces between subsystems. As you can see from the lower half of Figure 8.3, this leads to simplified communication, enabling effective communication even in large projects. For those same reasons, you should have one integration team.

**Configuration Management**

**Configuration Management System**

The Configuration Management (CM) system needs to be set up during Inception and refined in Elaboration when you baseline the architecture. Here, we describe why you need a CM system and
how it benefits you during Construction.

As a project progresses, it becomes increasingly complex to track all versions of the many files being created and changed. Especially during iterative development, you are continually creating new versions of files, and you need help with, among others, the following tasks:

- Iterative development means frequent builds, maybe even daily builds. You need to track which version goes into each build. Sometimes it is the latest version, sometimes an earlier version because the work on a new version has not been completed.

- Iterative development means that you will try out different solutions to see if you can improve upon what you have. You need to merge successful solutions into your mainstream development paths.

- As a project grows, it is essential to be able to hide changes made by one team from other teams so they will not be negatively impacted by code that is not yet working properly. Sometimes you want to make your work visible to the rest of the project several times a day, sometimes only every few days.

- For larger projects, you may also want to control who is allowed to make certain types of changes.\[1\]

\[1\] Note that even though some industry leaders propose that anybody should be able to change any code or design in the system, our experience shows that this practice does not scale effectively as projects grow beyond 10 people.

- When you notice that you have introduced a defect, you want to be able to go back in time to understand when the defect was introduced. You may also need to be able to go back to old builds quickly to get hold of a build that works (for example, if last night's updates to your e-commerce site made your site crash).

The solution is to use a CM system that tracks all this. You need to set up "units of responsibility/visibility"\[2\] attributes in your CM system. A robust architecture helps here since "units of responsibility/visibility" are typically mapped to the subsystem structure. It is the configuration manager's responsibility to set up the CM system.

\[2\] Rational ClearCase refers to these as Versioned Object Bases (VOBs).

Once the CM system has been set up, team members can pay less attention to the thousands of code files, documents, and their versions, allowing them to focus more on core development tasks, thus increasing their efficiency.

**Integration Planning**

With iterative development, it becomes increasingly complex to plan builds and integration testing. Each iteration needs an Integration Build Plan specifying which capabilities should be testable in each build and which components need to be integrated to produce required capabilities, such as use cases, parts of use cases, or other testable functionality. The tests may include functional, load, stress, or other types of tests.

In many cases, a build is produced by incrementally integrating several smaller builds. This is typically done bottom-up in the layered structure of subsystems in the implementation model. For each build, define which subsystems should go into it and which other subsystems must be
available as stubs (that is, as mock-up code emulating the required capabilities of the subsystem). Figure 8.4 shows three builds.

**Figure 8.4. Incremental Builds Facilitate Builds for Large Systems.** *Builds of large systems are often done by integrating several smaller builds, starting from the bottom. This figure shows an integration performed in three builds: build 1 and test, add build 2 and test, then add build 3 and test.*

![Diagram showing incremental builds for large systems]

**Enforce the Architecture**

To benefit fully from the work done with the architecture, you need to actively enforce the architecture. Small teams should be able to achieve this by frequently discussing design options. Larger teams need to pay more attention to this problem.

You have defined architectural mechanisms—that is, reusable solutions to common problems, such as dealing with persistency or inter-process communication. You now need to prevent each developer from arbitrarily reinventing solutions for these problems. This can be done through training on the architecture and architectural mechanisms available, coupled with design reviews that include the architects and developers.

You also need to ensure that interfaces of subsystems are not arbitrarily changed and that any changes are properly communicated to other teams to minimize impact on them. This communication can be done through a Configuration Management system, where you, among others, put interfaces under CM.

**Ensure Continual Progress**

To ensure continual progress, you need to establish short-term goals and continually prove that you have achieved them. The following guidelines are proven recipes for success:

- **Create one team, with one mission.** You want to avoid functionally oriented teams, where
analysts are organized in one team and throw the requirements over the wall to the developers who design and implement the requirements and in turn throw the code over the wall to the poor testers, who try to figure out what should be tested. Instead, you should have cross-functional teams, where each team member feels responsible for the application and for the team making progress. Having a quick daily meeting in which the team discusses status and decides what to focus on next can also facilitate team building.\[3\]

- **Set clear, achievable goals for developers.** Each developer should have a very clear picture of what to accomplish in a given iteration, if not within a portion of the iteration. The developers should agree that the expected deliverables are achievable.

- **Continually demonstrate and test code.** Measure progress primarily by looking at what code is compiled and tested. Do not be satisfied with statements such as "we are 90 percent done." Continual demonstration and testing of executable code is the only way to ensure progress.

- **Force continuous integration.** If possible, do daily builds. For large projects, or projects with poor or nonexistent CM systems, daily builds may not be feasible. Performing frequent builds ensures frequent integration testing, which provides feedback on the recent code that has been written since the last build. Continuous integration typically also reduces gold plating.

For each of our three example projects, you do the following:

- Project Ganymede, a small green-field project: As a major component was identified (already in Elaboration), the team decided who should be the main person responsible for its design, implementation, and testing. Team members frequently get together to discuss architectural issues and to review each other's design and implementations. They set up a CM system in Elaboration, which they used to do daily builds.

- Project Mars, a large green-field project: The development group is organized as three teams, each with the responsibility for one or two of the major subsystems. Within the teams, it is also clear which person is responsible for the design, implementation, and testing of each component. The teams have weekly architecture meetings in which they discuss architectural issues, and the architect is consistently involved in reviewing the work of the various teams, resolving issues as they arise. They had set up a CM system in Inception, which they used to do biweekly builds.

- Project Jupiter, a second generation of a large project: The project staff is divided into three teams, each with the responsibility for one or two of the major subsystems that were added or had major changes made to it. Within the teams, it is also clear which person is responsible for the design, implementation, and testing of each component. The architect is consistently involved in reviewing the work of the various teams, resolving issues as they arise. Since they are building upon a stable architecture, there are few major architectural issues to resolve. A CM system was already in place when they started the project, which
they used to do biweekly builds.

Objective 2: Iteratively Develop a Complete Product That Is Ready to Transition to Its User Community

Describe the Remaining Use Cases and Other Requirements

As you implement and test a use case, you often need to revisit at least some of the detailed requirements, and in many cases, you may even want to rethink the entire use case as you come up with better solutions. If the analyst and developer are different people, they need to discuss jointly how to interpret requirements if they are unclear or can be improved. Analysts should have a very good understanding of the business needs, but may have problems coming up with an optimal solution (they may be blind to how business is done today). Developers can in many cases come in with fresh eyes and find new, innovative ways to address identified business solutions. To build really great applications, it is important that there is an open dialog between analysts and developers to avoid losing the fresh perspective developers may provide.

Nonessential use cases and those with no major architectural impact are generally skipped in Elaboration. For example, if a general print feature has already been implemented and there is a use case for maintaining certain information, you can be fairly certain that adding a use case to print that information will not significantly impact on the architecture. Also, in some systems there are many similar use cases, having the same general sort of functionality, but for different entities or different actors, with different user interfaces. These types of use cases are often left to be detailed in Construction, along with partially detailed use cases—those use cases that have been detailed for the main flow, or a few, but not all, flows of events.

Many nonfunctional requirements, such as performance requirements or requirements around application stability, are essential to getting the architecture right, and most of them should have been properly documented by the end of Elaboration. You may, however, need to add to or detail some of these as you learn more about the system.

Fill in the Design

In Elaboration, you defined the subsystems and their interfaces, key components and their interfaces, and architectural mechanisms. If you have a layered architecture, you implemented or acquired the hard part of the lower layers—the infrastructure—and the architecturally significant use cases.

For each iteration in Construction, focus on completing the design of a set of components and subsystems and a set of use cases. For more information on use-case design, see Design Use-Case Realizations and Components in Chapter 17. As you implement components (consisting primarily of interfaces and stubs), you will see the need to create additional supporting components as a result of better understanding the system. In the earlier Construction iterations, focus on addressing the highest risks, such as those associated with interfaces, performance, requirements, and usability. Do this by designing, implementing, and testing only the most essential scenarios for your selected use cases. In later Construction iterations, focus on completeness until you eventually design, implement, and test all scenarios of the selected use cases.
Design the Database

During Elaboration, you made a first-draft implementation of the database. In the Construction phase, additional columns may be added to tables, views may be created to support query and reporting requirements, and indexes may be created to optimize performance, but major restructuring of tables should not occur (this would be a sign that the architecture was not stabilized and that the start of the Construction phase was premature).

Implement and Unit-Test Code

Iteration planning is primarily determined by deciding which use cases to implement and test, and when. Use-case implementation is done component-by-component. Generally, by the time you get to Construction, some of the components have already been implemented or partially implemented. And for layered system architectures, most of the components in the lower layers already are implemented. Figure 8.5 shows how the component implementations generally evolve over time.

**Figure 8.5. Evolution of Components over Time.** As time progresses, components become more and more complete, with lower layer components being finished more rapidly. Some higher layer components need to be implemented to drive requirements down to lower layers and to enable the effective testing of lower layer components.

Developers need to test their implementations continuously to verify that they behave as expected. To test component(s), you may need to design and implement test drivers and test stubs that emulate other components that will interact with the component(s). A visual modeling tool may be able to generate these test drivers and stubs automatically. Once you have the stubs, you can run a number of test scenarios. Usually, test scenarios are derived from the use-case scenarios in which the component(s) participate, since the use-case scenarios identify how the components will
interact when the users are running the application. You also look at the nonfunctional requirements to understand any other constraints that need to be tested.

**Do Integration and System Testing**

When producing a build, components are integrated in the order specified in the integration build plan. Usually, the build is subjected to a minimal integration test by the integration team before being fully tested.

To increase quality, continuously integrate and test your system. To minimize testing costs, you need to automate regression testing so you can run hundreds or thousands of regression tests daily or weekly toward the current build, thereby ensuring that newly introduced defects are rapidly found. The following steps will help you in your testing effort:

- Identify the targets of testing by analyzing the iteration plan to make sure that you properly test what is produced in the current iteration.

- Identify testing ideas: an enumerated list of ideas identifying potentially useful tests to conduct. Testing ideas are identified from sources including the risk list, change requests, use cases, other requirements artifacts, UML models, and so on.

- Analyze the testing ideas and select a subset from which to produce test cases. Define inputs, outputs, execution conditions, and points of observation and control. By analyzing the sum of test cases, you identify the overall test automation architecture, including the overall structure of essential test components and test scripts. You also identify how tests (derived from test cases) should be structured into test suites.

- Implement tests (manual or automated) for each test case. Organize tests into test suites, and then execute them.

- Analyze test failures, and then file defects and change requests.

See [Chapter 18](#) for more information on testing.

**Early Deployments and Feedback Loops**

Performing frequent builds forces continuous integration and verification that the code works. Integration and system testing also reveals many quality issues. Additionally, it is crucial to get early feedback on whether the application is useful and provides desired behavior, by exposing it to actual users. For example, maybe it is performing according to requirements, but the requirements do not quite make sense. This is especially important when developing unprecedented applications or applications in unfamiliar domains, where it is difficult to assess what the real requirements are.

Future users of the system often do not want to, or have the ability to, spend time on early versions of the application. It may, for example, be hard to convince any one user to spend time on providing you with feedback, since the benefits may not be obvious to the users. This is often the case when building commercial products, when the identity of future users is unknown. During early stages of Construction, the application may be hard to use, cumbersome to install, and filled with workarounds, so much so that it is difficult to put it in the hands of the target user group without active hand-holding.

Based on your needs for feedback and the availability of customers to provide it, you should choose
the right approach for getting feedback, which provides value both to the development team and to the future users of the system. These approaches include

- Bringing a few users to your development environment and demonstrating key capabilities.
- Bringing a few users to your development environment and having them use the product for some time.
- Installing the software at a test site and sitting with the users as they are using the software.
- For hosted applications, providing some users with early access. You probably need to guide the user through the application, which may not be stable or intuitive to use at this stage.

Typical results of successful early deployments and feedback loops include verification of whether requirements are right or need to be modified, feedback on usability and performance, and identification of insufficient capabilities.

Testing in a development environment that is not equivalent to the target (production) environment may produce misleading results. Organizations that focus on tight quality control may need to invest in a separate environment that is equivalent to that of the target environment. This simulated environment enables frequent test builds and more accurate test results.

**Prepare for Beta Deployment**

A beta deployment is "prerelease" testing in which a sampling of the intended audience tries out the product. Beta deployment is done at the end of the Construction phase and is the primary focus of the Transition phase. A successful beta program needs to be prepared in Construction.

Beta testing serves two purposes: First, it tests the application through a controlled actual implementation, and second, it provides a preview of the upcoming release. The deployment manager needs to manage the product's beta test program to ensure that both of these purposes are served.

It is important to get a good sampling of the intended audience by making sure that you have both novice and experienced users and users in different environments and with different needs. This variety will help ensure that all aspects of the products are properly tested.

It is also essential that the product is complete, based on the scope management that has occurred during the iterations. Although all features should be implemented, it is acceptable to have some unresolved quality issues, such as an unstable element (as long as it does not cause data loss), or Help files or dialog boxes with less than optimal crisness in their guidance, or partial implementation of a rarely used function. You need to include installation instructions, user manuals, tutorials, and training material, or you will not get feedback on them from the beta testers. The supporting material is essential, but unfortunately it often is not included.

**Prepare for Final Deployment**

For many projects, you need to prepare for the final deployment in Construction (and sometimes earlier, during Elaboration). These activities typically include

1. **Producing material for training users and maintainers to achieve user self-reliability later.**
2. **Preparing deployment site and converting operational databases.** To get the new system up and running, you may have to purchase new hardware, add space for new hardware, or convert data from earlier systems to the new system.

3. **Preparing for launch: packaging and production; preparing for rollout to marketing, distribution, and sales forces; preparing for field personnel training.** Especially when developing a commercial product, this range of activities should take place to ensure a successful launch.

We describe these and other activities related to the final deployment in more detail in Chapter 9.

For our three example projects, the work in Construction is not significantly different, with respect to coding, integration, and testing, except for the number of people involved. There are, however, differences in the activities related to deployment, which are outlined here:

- **Project Ganymede, a small green-field project:** The team works closely with the future users of the system and spends a fair amount of time demonstrating use cases to customers as they complete and test them; this interaction gives the team many ideas for how to improve the application. Since there would only be a handful of users of the system and the development team would be actively involved in the rollout and maintenance of the application, they spend almost no time on preparing for a beta or final release.

- **Project Mars, a large green-field project:** Since the product would be rolled out to the 37 offices worldwide, it is important to get input from a variety of offices. The team had identified eight people on three different continents representing large and small offices and who are treated as an extended part of the development team. Monthly, they walk through the progress that has been made and demonstrate new capabilities. This process provides valuable feedback, specifically in understanding how different the needs are for different offices in different countries. Toward the end of Construction, they also arrange for a week of training for the extended team so they get the skills to run a local beta program in each of their eight offices.

- **Project Jupiter, a second generation of a large project:** Since this is a commercial product, they need to interact closely with sales to identify a few key customers who would be willing to do a beta rollout of the product. Some of these beta customers were exposed to an internal version, which was facilitated by development team members who visited onsite to install the alpha release and help them get started. This allowed for some early and valuable feedback.

---

**Project Review: Initial Operational Capability Milestone**

The Construction phase ends with an important project milestone, the **Initial Operational Capability Milestone**, which is used to determine whether the product is ready to be deployed.
into a beta test environment by answering (among others) the following questions:

- Is this product release stable and mature enough to be deployed in the user community?
- Are all the stakeholders ready for the transition into the user community?
- Are actual resource expenditures versus planned expenditures still acceptable?

The Transition phase may have to be postponed by adding an iteration to Construction if the project fails to reach this milestone.

**Conclusion**

During Construction, the third of the four phases of the RUP, you make a lot of progress compared to where you were at end of Elaboration. In a successful Construction phase, you accomplish the following:

- **You develop software cost-effectively by taking advantage of the architectural baseline (including architectural mechanisms) created in Elaboration.** By aligning the organization in teams along the architecture, you achieve some degree of parallelism in the work of development teams.

- **You are able to scale up the project** to include more team members as necessary, by having a robust architecture baseline as a starting point and by organizing around the architecture.

- **You build and assess several internal releases** to ensure that the system is usable and addresses the user needs.

- **You move from an executable architecture to the first operational version of your system.** You have a fully functional beta version of the systems, including installation, supporting documentation, and training material.

**Chapter 9. The Transition Phase**

In this chapter, we introduce the fourth and final phase of the RUP lifecycle, the Transition Phase (see Figure 9.1). While you read this chapter, remember that you need to decide how formally you want to capture artifacts: This could be done in your head, on a whiteboard, or in a model or document.

**Figure 9.1. The Transition Phase. The Transition phase is the fourth and**
last phase of the RUP lifecycle; it has a well-defined set of objectives and is concluded by the Product Release Milestone. Use these objectives to help you decide which activities to carry out and which artifacts to produce.

You ended the Construction phase with the deployment of a fully functional beta version of the system, including the installer, supporting documentation, and training material. But you know that this beta version is not the final product and that it still requires fine-tuning of functionality, performance, and overall quality.

The focus of the Transition phase is to ensure that the software fully addresses the needs of its users. The Transition phase normally spans one or two iterations that include testing the product in preparation for release and making minor adjustments based on user feedback. At this point in the lifecycle, all major structural issues should have been worked out, and user feedback should focus mainly on fine-tuning, configuration, installation, and usability issues. More complex projects may have several iterations in Transition, with each iteration producing a more refined, deployable version of the system. Often in Transition you may have to complete some feature that had been postponed to meet some earlier deadline.

It should be noted that the Transition phase in the RUP approach differs radically from traditional development, primarily because you enter the phase with a reasonably stable, integrated, and tested version of the system. Conversely, in the traditional waterfall approach, the final integration phase often starts with major breakage—sometimes you cannot even compile the whole system, interfaces between subsystems are not compatible, or the system crashes frequently, resulting in major rework for your team and several weeks' delay before the system is up and running and ready to test. The introduced delays result in a large portion of management time being spent on resetting and renegotiating expectations from key stakeholders.

**Objectives of the Transition Phase**

The Transition phase has the following objectives:

1. **Beta test to validate that user expectations are met.** This typically requires some tuning activities such as bug fixing and making enhancements for performance and usability.

2. **Train users and maintainers to achieve user self-reliability.** These activities ensure that the adopting organization(s) are qualified to use the system and have moved any necessary data from earlier systems or taken any other measures required to operate the
new system successfully.

3. **Prepare deployment site and convert operational databases.** To get the new system up and running, you may have to purchase new hardware, add space for new hardware, or convert data from earlier systems to the new system.

4. **Prepare for launch-packaging, production, and marketing rollout; release to distribution and sales forces; field personnel training.** Especially when developing a commercial product, these activities should take place to ensure a successful launch.

5. **Achieve stakeholder concurrence that deployment baselines are complete and consistent with the evaluation criteria of the vision.**

6. **Improve future project performance through lessons learned.** This includes documenting lessons learned and improving the process and tool environment.

**Transition Iterations and Development Cycles**

**Transition and Iterations**

The Transition phase ranges from being very straightforward to extremely complex, depending on the kind of product (see Figure 9.2). A new release of an existing desktop product may be very simple, merely requiring some minor bug fixes that can be done in one iteration. The replacement of an air-traffic control system may be extremely complex, however, requiring several iterations in which additional features and integrations with other systems need to be added, and complex cutover activities need to take place to transition the business from using the old system to the new system. This transition can include running old and new systems in parallel, migrating data, training users, and adjusting business processes.

**Figure 9.2. Number of Iterations in Transition.** The number of iterations required in Transition varies: For a simple system requiring primarily minor bug fixing, one iteration may suffice. For a complex system, such as an air-traffic control system, several iterations may be required to add features and perform complex cutover activities to transition the business from using the old system to using the new system.
Activities performed in Transition iterations depend on the project goal(s). Most projects have a reasonably simple Transition, working on fixing bugs; the primary focus is implementation and testing. Occasionally, new features may have to be added, and the iteration is then similar to a Construction phase iteration in that it requires some work on requirements, analysis and design, and so on.

When developing commercial products, and sometimes applications for wide internal use, you have to address packaging, production, marketing, and potential rollout to a sales and support organization. This is described in more detail in the section Objective 4: Prepare for Launch: Packaging, Production, and Marketing Rollout.

When developing applications that require a lot of new hardware, such as large financial processing systems, or require conversion of data from old to new systems, you have to address the activities described in the section Objective 3: Prepare Deployment Site and Convert Operational Databases.

Very complex projects may require an incremental delivery approach, each deployment providing an increasingly more complete and higher quality version of the system. This approach may be necessary if the only way to fine-tune the system is with feedback from actual system usage. This method is common for large-scale management information systems, or command and control systems, with distributed deployment and complex hardware requiring multiple systems to be integrated and fine-tuned together. Transition iterations for such projects would look very much like the final iteration or final two iterations in Construction (see Chapter 8), with the added
complexity of having to manage multiple deployments. Describing the complexity of these types of projects is outside the scope of this book.

**Transition and Development Cycles**

By the end of Transition, the phase objectives should have been met, and the project should be in a position to be closed. For some projects, the end of the current lifecycle may coincide with the start of another lifecycle, leading to the next generation of the same product. For other projects, the end of Transition may coincide with a complete delivery of the artifacts to a third party, who may be responsible for operations, maintenance, and enhancements of the delivered system.

One pass through the four RUP phases (Inception, Elaboration, Construction, and Transition) is a **Development Cycle**: At the end of Transition you have completed a development cycle (see **Figure 9.3**). Each development cycle produces a **generation** of the software. Unless the product "dies," it will evolve into its next generation by repeating the same sequence of Inception, Elaboration, Construction, and Transition phases, but with a different emphasis on the various phases, as required for the new project goals. These subsequent cycles are called **evolution cycles** (see also **Chapter 12**).

**Figure 9.3. Development Cycles. A Development Cycle consists of one pass through the four RUP phases, usually broken down into five to nine iterations, and followed by another Development Cycle (evolution cycle). In many cases, the Inception phase of the evolution cycle overlaps with the Transition phase of the existing cycle.**

There is often an overlap between two development cycles, where the Inception phase of the next development cycle overlaps with the Transition phase of the existing one. You may choose not to have an overlap or to have a bigger overlap, but by starting the next development cycle much earlier than the current Transition phase, you introduce a substantial amount of complexity, requiring mature development organizations with advanced Configuration Management processes.

Whereas major evolution of a system should be undertaken as a project using a RUP lifecycle and its four phases, many systems simply enter into another mode in which operations, support, and routine maintenance are provided on an ongoing basis, mostly driven by external requests and not with any predefined phases and milestones. A bug-fix release can take the shape of a very simple iteration, similar to what you would do in Transition, running through a few of the RUP activities in implementation, testing, and deployment.[1]

[1] See **Kruchten 2000b**.
Objective 1: Beta Test to Validate That User Expectations Are Met

Capturing, Analyzing, and Implementing Change Requests

The first operational version—the beta release—of the system was completed and deployed by end of Construction. During Transition you do the beta testing, which provides a lot of user feedback from the beta testers. To ensure that the feedback is useful, actively gather it through interviews, online queries, submitted change requests, or other means. You need to take the time to analyze the feedback, and submit and review change requests so you understand what changes are required before final product release.

Change requests (mainly defects and beta test feedback) are the major planning input for continuing development. Mostly, the change requests are only for minor system tweaking, such as fixing minor bugs, enhancing documentation or training material, or tuning the performance. Sometimes additional features must be added, requiring that you work with requirements, analysis and design, implementation, and testing (see the section Iteratively Develop a Complete Product That Is Ready to Transition to Its User Community, in Chapter 8).

It should be noted that adding features this late in the project could indicate failure in earlier phases, but could be well-motivated especially for very large and complex systems. In most cases you should refrain from adding new features and instead postpone them to the next development (evolution) cycle. In some cases, however, the system may not deploy successfully without the additional feature.

As you implement change requests, it is essential to have in place a good Configuration Management process as well as comprehensive regression testing. At this stage, builds with incorrect file versions or missing files are common sources of defects. Good CM practices and tools radically reduce these types of errors, and comprehensive regression testing rapidly identifies any introduced defects.

During Transition, you need to spend a fair amount of time improving documentation, online Help, training material, user's guides, operational guides, and other supporting documentation. It is important that actual beta testers properly test these elements in the target environment.

Transition Testing

The focus of testing during Transition shifts toward improving quality and avoiding regression. Additionally, there is often a requirement for formal acceptance testing, which may involve a repeat of all or part of the system-level tests. In planning for Transition testing, you should provide effort and resources for the following:

- **Continued test design and implementation** to support ongoing development.
- **Regression testing**, which will require variable effort and resources, depending on the chosen approach; for example, retest everything or retest to an operational profile.
- **Acceptance testing**, which may not require the development of new tests.

As defects are fixed and beta feedback is incorporated, successive builds are tested using a standard test cycle:

- **Validate build stability.** Execute a subset of tests to validate that the build is stable enough to start detailed test and evaluation.

- **Test and evaluate.** Implement, execute, and evaluate tests.

- **Achieve your test objectives, or, in RUP parlance, achieve an acceptable mission.** Evaluate test results against testing objectives and perform additional testing as necessary.

- **Improve test assets.** Improve test artifacts as needed to support the next cycle of testing.

When the system is deemed fit to undergo acceptance testing, a separate testing cycle is performed focusing on executing tests and evaluating results.

**Patch Releases and Additional Beta Releases**

If serious defects that prevent effective beta testing are found, you may need to produce a patch release (special bug-fix release installed on top of the current baselined release) and make it available to beta customers for download. Often these patch releases may be available only to specific beta testers to address their special areas of concern or to block defects they are encountering (although sometimes they go to all beta testers).

As described in an earlier section, Transition and Iterations, there may be more than one iteration in Transition. If so, you typically produce a new release that is distributed to all beta testers, such as Beta 1, Beta 2, and so on. You need to ensure that enough beta testers are upgrading to the new release to provide sufficient feedback.

**Metrics for Understanding When Transition Will Be Complete**

It is not always obvious when you can end Transition. To help determine when Transition will be complete, you can analyze defect metrics and test metrics, among other things. This analysis can help you answer questions such as, "When will the quality be good enough?", "How many more defects can we expect to find?", and "When will we have tested all functionality?"

Let's look at a small sample set of metrics you may find useful: defect metrics and test metrics.

**Defect Metrics**

For defects, you should track

- How many new defects are **found** each day.

- How many defects are **fixed** each day.

**The important thing is to focus on the trend rather than the actual number of defects.** Figure 9.4 shows that the number of open defects increased until February 10, and as you moved more and more developers to fix defects, the code improved in quality. You closed more defects
than you opened, leading to a rapid decrease in the number of open defects. Through trend analysis of this figure, you can estimate that 20 open defects will be reached around March 9, and another week would be needed to come down to below 5 defects. However, it is common that as you approach 0 open defects, the defect-fixing rate will slow down—this needs to be factored into your estimates.

**Figure 9.4. Trend Analysis of Defects.** By analyzing a defect trend, you can predict when a certain threshold value of open defects will be reached. If you plan to release the product once there are less than 20 defects, the graph shows that this will likely occur around March 9.

![Defect Trends in Transition](image)

**Test Metrics**

Another important set of metrics of great value is test metrics, such as test completion. If, for example, only 60 percent of planned tests are completed, you should expect to find many more defects once testing is completed. You can predict how many new defects can be expected by determining how many defects are typically found per test case and multiplying that by the number of test cases yet to be executed and analyzed. This can be expressed with the formula:

\[
D_{\text{Remaining}} = D_{\text{Average}} \times (T_{\text{Total}} - T_{\text{Executed}})
\]

*Where*

- \(D_{\text{Remaining}}\) = number of defects yet to be found
- \(D_{\text{Average}}\) = average number of defects found per test case
- \(T_{\text{Total}}\) = total number of test cases
- \(T_{\text{Executed}}\) = number of test cases executed so far

You do not want to deliver a system that has not been properly tested. By analyzing the trend of
completed and successful test cases, you can predict when you will have completed the test effort, yet another data point allowing you to decide when it is appropriate to end Transition.

Objective 2: Train Users and Maintainers to Achieve User Self-Reliability

During Transition, make sure that all users, operational staff, and maintenance teams are given appropriate training. This training also allows you to do advanced beta testing of training material, user documentation, and operational manuals.

When you need to train a large number of users, such as when rolling out a new system enterprise wide, or you need high-quality training material, such as for many commercial products, you typically need to start this activity much earlier, in Construction (or often in Elaboration). You need to develop training material and train your instructors during Construction (and possibly have already started your planning in Elaboration). In conjunction with the launch of the beta release, you need to do test runs of the training sessions to make sure that the training does the job. For a commercial product, or other rollouts where you have to bring a large number of instructors up to speed on the material, you may need to run a number of "Train the Trainer" sessions in conjunction with the beta release.

Objective 3: Prepare Deployment Site and Convert Operational Databases

Making a smooth transition may be very complex when replacing an existing system. Data needs to be transferred, and the new and old systems may need to be run in parallel for some time to ensure accurate performance of the new system. This may lead to a lot of extra work in terms of entering information into both systems and verifying that the system functions match.

In some cases, you may need more room or bigger facilities to house new machines, you may need to arrange for a power supply or backup power, you may need to install networks and systems for backup, and so on. Turnkey systems may require rollout onto various desktops and servers.

It should be noted that for many complex deployments, you may need to start these activities much earlier (typically in Elaboration or in Construction).

Objective 4: Prepare for Launch: Packaging, Production, and Marketing Rollout

Although this section is primarily relevant for independent software vendors developing shrink-wrapped products, some of the activities that follow are useful for other developments, especially
for applications rolled out to a large set of internal customers.

It should be noted that for many large rollouts of commercial products, you may need to start this activity much earlier (in Elaboration or in Construction).

**Packaging, Bill of Materials, and Production**

The Bill of Materials (BOM) uniquely identifies all the constituent parts of a product. The final packaged product will consist of the software on some storage media, some documents or manuals, licensing agreement forms, and the packaging itself.

You need to ensure that all the items for manufacturing are in their final approved state at the time of delivery to the manufacturer. The approved application and installation software will need to be checked for viruses and saved on a mass-producible storage medium (for example, a master CD). Manuals and printed materials need to be in "camera-ready" format. Once all the component parts are in place and complete, they can be handed over to the manufacturing organization for mass production and duplication.

**Marketing Rollout**

If you are building a commercial product, here are some of the artifacts you might want to consider:

- **Core Message Platform (CMP).** A one- to two-page description providing short, medium, and long descriptions of the product, its positioning, and key features and benefits. The CMP is the cornerstone in any successful launch and is used as a baseline or template for all internal and external communication related to the product.

- **Customer-consumable collateral.** Data sheets, whitepapers, technical papers, information on your Web site, prerecorded demos of the product, demo scripts, and multimedia presentations providing an overview of the product.

- **Sales support material.** Sales presentations, technical presentations, field training material, fact sheets, positioning papers, competitive write-ups, coaching on how to meet sales objections, references, success stories, and so on.

- **Launch material.** Press releases, press kits, analyst briefings, and internal newsletters.

**Objective 5: Achieve Stakeholder Concurrence That Deployment Is Complete**

**Product Acceptance Test**

Product acceptance testing is the final test action prior to deploying the software. The goal of acceptance testing is to verify that the software is ready and can perform those functions and tasks it was built for.

There are three common strategies for acceptance test:
- Formal acceptance
- Informal acceptance
- Beta test

**Formal acceptance** testing is a highly managed process where there is a clear one-to-one supplier-acquirer relationship. It is often an extension of the system test. The tests are planned and designed carefully and in the same detail as system testing. The test cases selected should be a subset of those performed in system testing. It is important not to deviate in any way from the chosen test cases. In many organizations, formal acceptance testing is fully automated. Formal acceptance testing is most frequently performed by the supplying organization under control of the acquirer, by the acquirer itself, or by a third party appointed by the acquirer.

**Informal acceptance** testing has less rigorously defined test procedures than those of formal acceptance testing. The functions and business tasks to be explored are identified and documented, but there are no particular test cases to follow. The individual tester determines what to do. This approach to acceptance testing is not as controlled as formal testing and is more subjective than the formal one. Informal acceptance testing is most frequently performed by the end-user organization.

**Beta testing** is described in an earlier section, Objective 1: Beta Test to Validate That User Expectations Are Met.

Independent of the approach selected, you should agree on the test cases and how they should be evaluated before acceptance testing is implemented and executed.

Product acceptance testing often involves more than executing the software for readiness; it also involves all product artifacts delivered to the customer(s), such as training, documentation, and packaging. Evaluating the non-software artifact(s) varies greatly depending on the artifact being evaluated (refer to the Guidelines and Checklists, available in the RUP product, for information regarding what and how to evaluate). For examples, see the Guidelines and Checklists for the artifacts Use Case and Software Architecture Document, in the RUP product.

**Objective 6: Improve Future Project Performance Through Lessons Learned**

At the end of each project, it is advisable to spend some time analyzing and documenting what worked well and what didn’t. What refinements would you recommend for your process or development environment? What are other lessons learned? This is often done in a post-mortem assessment. Based on the results, you update your RUP development case and improve your development environment to reflect the lessons learned.

You should also determine whether any work can be reused for other projects. If so, make sure to remove any sensitive data and store the reusable assets in such a way that they can easily be found and used by other teams.
Project Review: Product Release Milestone

Transition ends with the fourth major project milestone, the Product Release Milestone, to determine whether the objectives were met and if you should start another development cycle. (Several development cycles may have been already planned during Inception.) In some cases this milestone may coincide with the end of the Inception phase for the next cycle. The primary evaluation criteria for Transition answers these questions:

- Are the users satisfied?
- Are actual resource expenditures versus planned expenditures acceptable, and, if not, what actions can be taken in future projects to address this issue?

At the Product Release Milestone, the product is in production, and a process of maintenance, operation, and support begins. This may involve starting a new development cycle with new major enhancements or some additional maintenance release to fix found defects.

Conclusion

During Transition, the fourth and last of the RUP lifecycle phases, you ensured that the software addresses the needs of its users and can be successfully deployed in the target environment. You met the following objectives:

- You performed one or more beta tests of the new system with a small set of actual users and fine-tuned it as necessary.
- You trained users and maintainers to make them self-reliant.
- You prepared the site for deployment, converted operational databases, and took other measures required to operate the new system successfully.
- You launched the system with attention to packaging and production; rollout to marketing, distribution, and sales forces; and field personnel training. This is specifically a focus for commercial products.
- You achieved stakeholder concurrence that deployment baselines are complete and consistent with the evaluation criteria of the vision.
- You analyzed and used lessons learned to improve future project performance.