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.

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 time?).

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.
OBJECTIVES OF THE ELABORATION PHASE

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

To validate an architecture, you need more than a review of a paper representation; you need an executable architecture that can be tested.

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
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\(^1\) 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.

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,\(^2\) 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.

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

---

1. A subsystem corresponds to a component or a collection of components.
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.
Chapter 7 ▼ The Elaboration Phase

Architecturally significant use cases

Equipment- and customer-specific code

Processes and other application code

Major abstractions, classes, etc.

Mechanisms, services (e.g., ORB, MQS,...)

H/W-specific code, O/S-specific code, general-purpose code

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.

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. 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.

---

**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.
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.

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.

- **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.

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

---

**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.
DESIGN, IMPLEMENT, VALIDATE, AND BASELINE THE ARCHITECTURE

cart” implementation, or communication with external systems through a certain protocol.

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 Incep-
tion. 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?
CHAPTER 7  ▼  THE ELABORATION PHASE

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.