Chapter 2

SPE Quick View

The aim of science is not to open the door to infinite wisdom, but to set a limit to infinite error.

—Bertolt Brecht

In This Chapter:

- The SPE process
- Example illustrating the process
- SPE in the Unified Software Process
- Creating software with good performance characteristics

2.1 SPE Process for Object-Oriented Systems

For object-oriented systems, we adapt the general SPE techniques to the process typically followed for object-oriented development, and to the artifacts that it produces.

The SPE process focuses on the system's use cases and the scenarios that describe them. In a use-case-driven process such as the Unified Process ([Kruchten 1999], [Jacobson et al. 1999]), use cases are defined as part of requirements definition (or earlier) and are refined throughout the design process. From a development perspective, use cases and their scenarios provide a means of understanding and documenting the system's requirements, architecture, and design. From a performance perspective, use cases allow you to identify the *workloads* that are significant from a performance point of view, that is, the collections of requests made by the users of the system. The scenarios allow you to derive the processing steps involved in each workload.

Chapter 15 describes these steps in more detail.

The SPE process includes the following steps. The activity diagram in Figure 2-1 captures the overall process.

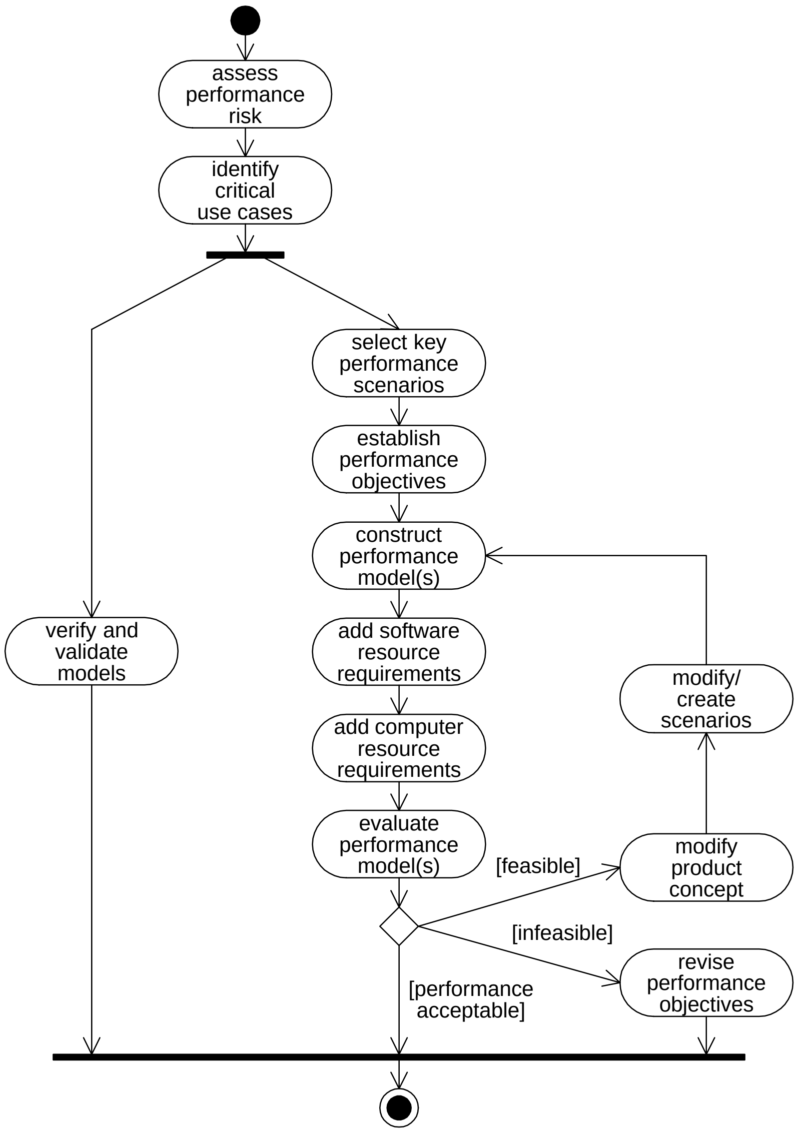
assess performance risk
identify critical use cases
select key performance scenarios
establish performance objectives
construct performance model(s)
add software resource requirements
add computer resource requirements
evaluate performance model(s)
verify and validate models
modify/ create scenarios
modify product concept
revise performance objectives
[feasible]
[infeasible]
[performance acceptable]

Figure 2-1: The SPE Process for Object-Oriented Systems

1. *Assess performance risk*: Assessing the performance risk at the outset of the project tells you how much effort to put into SPE activities. If the project is similar to others that you have built before, is not critical to your mission or economic survival, and has minimal computer and network usage, then the SPE effort can be minimal. If not, then a more significant SPE effort is needed.

Use cases are discussed in Chapter 3.

2. *Identify critical use cases*: The critical use cases are those that are important to the operation of the system, or that are important to responsiveness as seen by the user. The selection of critical use cases is also risk driven. You look for use cases where there is a risk that, if performance goals are not met, the system will fail or be less than successful.

   Typically, the critical use cases are only a subset of the use cases that are identified during object-oriented analysis. In the UML, use cases are represented by use case diagrams.

Sequence diagrams and their extensions are discussed in Chapter 3.

3. *Select key performance scenarios*: It is unlikely that all of the scenarios for each critical use case will be important from a performance perspective. For each critical use case, the key performance scenarios are those that are executed frequently, or those that are critical to the perceived performance of the system. Each performance scenario corresponds to a workload. We represent scenarios by using sequence diagrams augmented with some useful extensions.

Performance objectives are discussed in Chapter 7.

4. *Establish performance objectives*: You should identify and define *performance objectives* and *workload intensities* for each scenario selected in step 2. Performance objectives specify the quantitative criteria for evaluating the performance characteristics of the system under development. These objectives may be expressed in three primary ways by response time, throughput, or constraints on resource usage. For information systems, response time is typically described from a user perspective, that is, the number of seconds required to respond to a user request. For real-time systems, response time is the amount of time required to respond to a given external event. Throughput requirements are specified as the number of transactions or events to be processed per unit of time.

   Workload intensities specify the level of usage for the scenario. They are specified as an arrival rate (e.g., number of Web site hits per hour) or number of concurrent users.

Repeat steps 5 through 8 until there are no outstanding performance problems.

Chapter 4 discusses execution graphs.

5. *Construct performance models*: We use execution graphs to represent software processing steps in the performance model. The sequence-diagram representations of the key performance scenarios are translated to execution graphs.

Gathering data on software and computer resource requirements is discussed in Chapter 7 and Part III.

6. *Determine software resource requirements:* The processing steps in an execution graph are typically described in terms of the software resources that they use. Software resource requirements capture computational needs that are meaningful from a software perspective. For example, we might specify the number of messages sent or the number of database accesses required in a processing step.

   You base estimates of the amount of processing required for each step in the execution graph on the operation specifications for each object involved. This information is part of the class definition in the class diagram. As described in Chapter 4, when done early in the development process, these may be simple best- and worst-case estimates. Later, as each class is elaborated, the estimates become more precise.

Resource requirements are discussed in Chapter 7.

7. *Add computer resource requirements:* Computer resource requirements map the software resource requirements from step 6 onto the amount of service they require from key devices in the execution environment. Computer resource requirements depend on the environment in which the software executes. Information about the environment is obtained from the UML deployment diagram and other documentation. An example of a computer resource requirement would be the number of CPU instructions and disk I/Os required for a database access.

Note: Steps 6 and 7 could be combined, and the amount of service required from key devices estimated directly from the operation specifications for the steps in the scenario. However, this is more difficult than estimating software resources in software-oriented terms and then mapping them onto the execution environment. In addition, this separation makes it easier to explore different execution environments in "what if" studies.

Performance models and their solutions are discussed in Chapters 4, 5 and 6.

8. *Evaluate the models:* Solving the execution graph characterizes the resource requirements of the proposed software alone. If this solution indicates that there are no problems, you can proceed to solve the system execution model. This characterizes the software's performance in the presence of factors that could cause contention for resources, such as other workloads or multiple users.

   If the model solution indicates that there are problems, there are two alternatives:
   - *Modify the product concept*: Modifying the product concept involves looking for feasible, cost-effective alternatives for satisfying this use case instance. If one is found, we modify the scenario(s) or create new ones and solve the model again to evaluate the effect of the changes on performance.
   - *Revise performance objectives*: If no feasible, cost-effective alternative exists, then we modify the performance goals to reflect this new reality.

     It may seem unfair to revise the performance objectives if you can't meet them (if you can't hit the target, redefine the target). It is not wrong if you do it at the outset of the project. Then all of the stakeholders in the system can decide if the new goals are acceptable. On the other hand, if you get to the end of the project, find that you didn't meet your goals, and *then* revise the objectives—*that's* wrong.

Model verification and validation is discussed in Chapter 15.

9. *Verify and validate the models*: Model verification and validation are ongoing activities that proceed in parallel with the construction and evaluation of the models. Model verification is aimed at determining whether the model predictions are an accurate reflection of the software's performance. It answers the question, "Are we building the model right?" For example, are the resource requirements that we have estimated reasonable?

   Model validation is concerned with determining whether the model accurately reflects the execution characteristics of the software. It answers the question [Boehm 1984], "Are we building the right model?" We want to ensure that the model faithfully represents the evolving system. Any model will only contain what we think to include. Therefore, it is particularly important to detect any model omissions as soon as possible.

Both verification and validation require measurement. In cases where performance is critical, it may be necessary to identify critical components, implement or prototype them early in the development process, and measure their performance characteristics. The model solutions help identify which components are critical.

Late life cycle and post deployment SPE activities are discussed in Chapter 15.

These steps describe the SPE process for one phase of the development cycle, and the steps repeat throughout the development process. At each phase, you refine the performance models based on your increased knowledge of details in the design. You may also revise analysis objectives to reflect the concerns that exist for that phase.

2.2 Case Study

To illustrate the process of modeling and evaluating the performance of an object-oriented design, we will use an example based on an automated teller machine (ATM). This example is based on a real-world development project in which one of the authors participated. It has been simplified for this presentation, and some details have been changed to preserve anonymity.

> The ATM accepts a bank card and requests a personal identification number (PIN) for user authentication. Customers can perform any of three transactions at the ATM: deposit cash to an account, withdraw cash from an account, or request the available balance in an account. A customer may perform several transactions during a single ATM session. The ATM communicates with a computer at the host bank, which verifies the customer-account combination and processes the transaction. When the customer is finished using the ATM, a receipt is printed for all transactions, and the customer's card is returned.

The following sections illustrate the application of the SPE process for object-oriented systems to the ATM.

2.2.1 Assess Performance Risk (Step 1)

The performance risk in constructing the ATM itself is small. Only one customer uses the machine at a time, and the available hardware is more than adequate for the task. Consequently, the amount of SPE effort on this project will be small. However, the host software (considered later) must deal with a number of concurrent ATM users, and response time there is important, so a more substantial SPE effort is justified.

2.2.2 Identify Critical Use Cases (Step 2)

We begin with the use case diagram for the ATM shown in Figure 2-2. As the diagram indicates, several use cases have been identified: Operator-Transaction (e.g., reloading a currency cassette), CustomerTransaction (e.g., a withdrawal), and CommandFunctions (e.g., to go off-line). Clearly, CustomerTransaction is the critical use case, the one that will most affect the customer's perception of the ATM's performance.

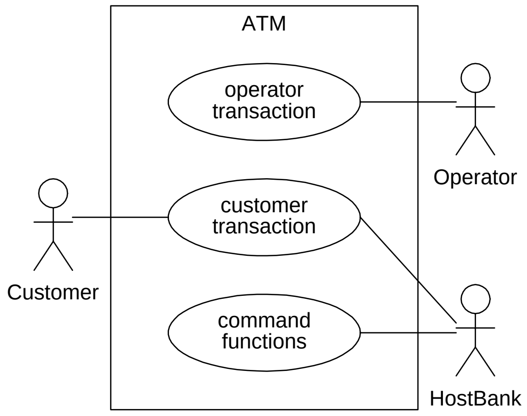

Figure 2-2: ATM Use Cases

2.2.3 Select Key Performance Scenarios (Step 3)

We therefore select the CustomerTransaction as the first performance scenario to consider. This scenario represents typical, error-free customer transactions from the CustomerTransaction use case. Later, after we confirm that the architecture and design are appropriate for this scenario, we will consider additional scenarios. To evaluate the scenario, we need a specification for the *workload intensity*—that is, the number of CustomerTransactions or their arrival rate during the peak period.

Chapter 3 discusses sequence diagrams and their extensions.

Figure 2-3 shows a scenario for customer transactions on the ATM. The notation used is a UML sequence diagram augmented with some additional features. These features allow us to denote repetition and choice. They are indicated by the rectangular areas labeled loop and alt, respectively. This scenario indicates that, after inserting a card and entering a PIN, a customer may repeatedly select transactions which may be deposits, withdrawals, or balance inquiries. The rounded rectangles indicate that the details of these transactions are elaborated in additional sequence diagrams.

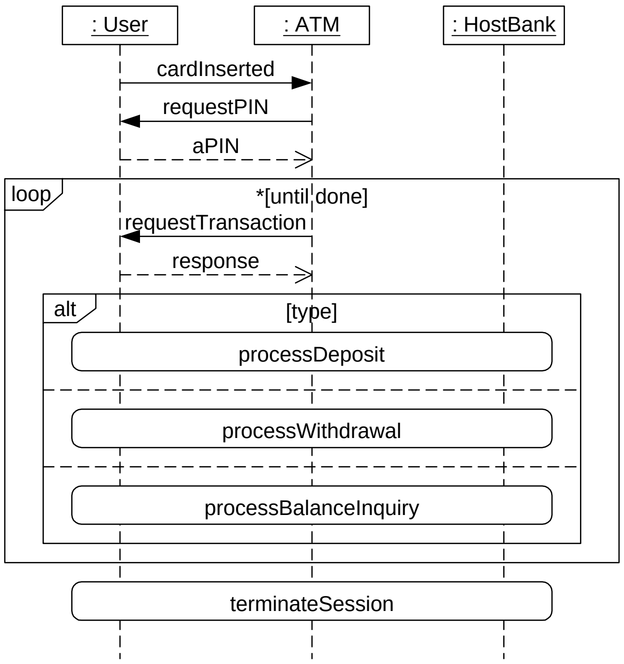

Figure 2-3: Customer Transaction Scenario

The scenario in Figure 2-3 combines the customer transactions of deposit, withdrawal, and balance inquiry. We combine them because we want to model what a customer does during an ATM session, and a customer may request more than one transaction during a single session. While we don't know exactly which transaction(s) a user will request, we can assign probabilities to each type of transaction based on reasonable guesses or actual measurements of customer activities.

Note: We could also represent the information in Figure 2-3 using a UML activity diagram. However, we have found that the extended sequence diagram notation is more familiar to software developers, and is easier to translate to a software execution model.

2.2.4 Establish Performance Objectives (Step 4)

As a bank customer, what response time do you expect from an ATM? Historically, performance objectives have been based on "time in the (black) box," that is, the time from the arrival of the (complete) request to the time the response leaves the host computer. That approach was used to separate things outside the control of the software (e.g., the time for the user to enter information, network congestion, and so on) from those that are more directly influenced by the software itself. If we take this approach for the ATM, a reasonable performance objective would be one second for the portion of the time on the host bank for each of the steps processDeposit, processWithdrawal, and processBalanceInquiry.

However, for SPE we prefer to expand the scope to cover the end-to-end time for a customer to complete a business task (e.g., an ATM session). Then, the results of the analysis will show opportunities to accomplish business tasks more quickly by reducing the number and type of interactions with the system, in addition to reducing the processing "in the box." A reasonable performance objective for this scenario might be 30 seconds or less to complete the (end-to-end) ATM session.

2.2.5 Construct Performance Models (Step 5)

The models for evaluating the performance of the ATM are based on the key scenarios identified earlier in the process. These *performance scenarios* represent the same processing as the sequence diagrams using execution graphs.

Chapter 4 discusses translating scenarios to execution graphs.

Figure 2-4 shows the execution graph that corresponds to the ATM scenario in Figure 2-3. The rectangles indicate processing steps; those with bars indicate that the processing step is expanded in a subgraph. Figure 2-5 shows the expansion of the processTransaction step; the expansion of the other steps is not shown here. The circular node indicates repetition, while the jagged node indicates choice.

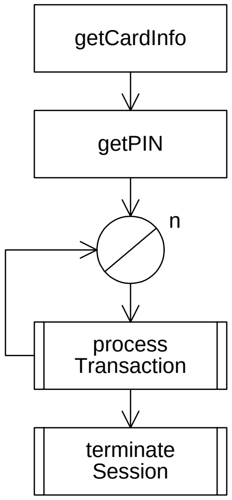

Figure 2-4: ATM Execution Graph

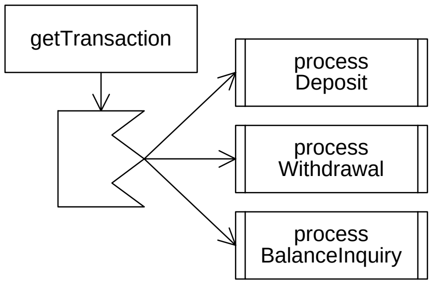

Figure 2-5: Expansion of `processTransaction`

The execution graph in Figure 2-4 expresses the same scenario as the sequence diagram in Figure 2-3: After inserting a card (to provide customer information) and entering a PIN, a customer may repeatedly select transactions, which may be deposits, withdrawals, or balance inquiries. Here, the number of transactions that a customer may perform is indicated by the parameter ***n***.

2.2.6 Determine Software Resource Requirements (Step 6)

Software resources are discussed in more detail in Chapters 4 and 7.

The types of software resources will differ depending on the type of application and the operating environment. The types of software resources that are important for the ATM are:

- `Screens`—the number of screens displayed to the ATM customer
- `Host`—the number of interactions with the host bank
- `Log`—the number of log entries on the ATM machine
- `Delay`—the relative delay in time for other ATM device processing, such as the cash dispenser or receipt printer

Note: Software resource requirements are application-technology specific. Different applications will specify requirements for different types of resources. For example, a system with a significant database component might specify a software resource called “DBAccesses” and specify the requirements in terms of the number of accesses. We cover the identification of applicable software resources in Part III.

We specify requirements for each of these resources for each processing step in the execution graph, as well as the probability of each case alternative and the number of loop repetitions. Figure 2-6 shows the software resource requirements for `processWithdrawal`.

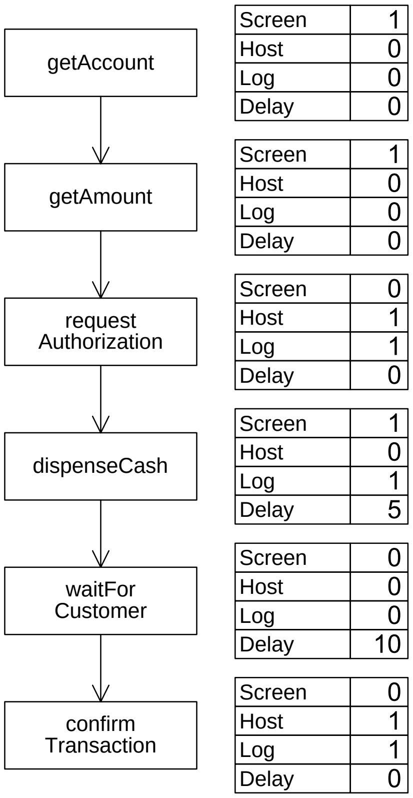

Figure 2-6: Software Resource Requirements for `processWithdrawal`

2.2.7 Add Computer Resource Requirements (Step 7)

We must also specify the *computer resource requirements* for each software resource request. The values specified for computer resource requirements connect the values for software resource requirements to device usage in the target environment. The computer resource requirements also specify characteristics of the operating environment, such as the types of processors/devices, how many of each, their speed, and so on.

Table 2-1: Example Overhead Matrix

Devices	CPU	Disk	Display	Delay	Net
Quantity	1	1	1	1	1
Service Units	Sec.	Phys. I/O	Screens	Units	Msgs.
Screen	0.001		1		
Host	0.005			3	2
Log	0.001	1			
Delay				1	
Service Time	1	0.02	1	1	0.05

Table 2-1 contains the computer resource requirements for the ATM example. The names of the devices in the ATM unit are in the first row, while the second row specifies how many devices of each type are in the facility. The third row is a comment that describes the unit of measure for the values specified for the software processing steps. The next four rows are the names of the software resources specified for each processing step, and the last row specifies the service time for the devices in the computer facility.

The values in the center section of the table define the connection between software resource requests and computer device usage. The `Display` "device" represents the time to display a screen and for the customer to respond to the prompt. The 1 in the `Display` column for the `Screen` row means that each `screen` specified in the software model causes one visit to the `Display` delay server. We arbitrarily assume this delay to be one second (in the service time row). Similarly, each `Host` and `Delay` specification in the software model results in a delay before processing the next step. We assume the `Host` delay is 3 seconds; other delays are specified in 1-second increments. Each `Host` request also sends 1 message via the `Net` and receives 1 reply message. Each message takes an average of 0.05 second. These values may be measured, or estimates could be obtained by constructing and evaluating more detailed models of the host processing required.

The connection between software resources and processing overhead is discussed in Chapter 4.

Thus, each value specified for a processing step in the software model generates a demand for service from one or more devices in a facility. The computer resource requirements define the devices used and the amount of service needed from each device. The demand is the product of the software model value times the value in the overhead matrix cell times the service time for the column.

2.2.8 Evaluate the Models (Step 8)

Details of the software execution model solution for the ATM are presented in Chapter 4.

We begin by solving the software model. This solution provides a "no contention" result. Here, we find that the total end-to-end time for the scenario is approximately 29 seconds, and most of that is due to the delays at the ATM unit for customer interactions and processing. This is a best-case result and, in this case, confirms that a single ATM will complete in the desired time. Because it is close to exceeding the objective, however, we may want to examine alternatives to reduce the end-to-end time. We should also examine the sensitivity of the results to our estimates. In addition, further studies will examine the host bank performance when there are multiple ATMs whose transactions could produce contention for computer resources. This will affect the time to handle `Host` requests.

2.2.9 Verify and Validate the Models (Step 9)

Chapter 6 presents details of the system execution model solution for the ATM.

Chapter 8 discusses performance measurement.

We need to confirm that the performance scenarios that we selected to model are critical to performance, and confirm the correctness of the workload intensity specifications, the software resource specifications, the computer resource specifications, and all other values that are input into the model. We also need to make sure that there are no large processing requirements that are omitted from the model. To do this, we will conduct measurement experiments on the operating environment, prototypes, and analogous or legacy systems early in the modeling process. We will measure evolving code as soon as viable. SPE suggests using early models to identify components critical to performance, and implementing them first. Measuring them and updating the model estimates with measured values increases precision in key areas early.

2.3 SPE in the Unified Software Process

To be effective, the SPE steps described in Section 2.2 should be an integral part of the way in which you approach software development. Integrating SPE into your software process avoids two problems that we have seen repeatedly. One is over-reliance on individuals. When you rely on individuals to perform certain tasks instead of making them part of the process, and then those individuals leave the company, their tasks are frequently forgotten. The other problem is that if SPE is not part of the process, it is easy to omit the SPE evaluations when time is tight or the budget is limited.

SPE deliverables are discussed in Chapter 15.

Integrating SPE into the software development process is not difficult. It is compatible with a wide variety of software process models, including the waterfall model [Royce 1970], the spiral model [Boehm 1988], and the Unified Process [Jacobson et al. 1999], [Kruchten 1999]. In each case, integrating SPE into your software process requires that you define the milestones and deliverables that are appropriate to your organization, project, and the level of SPE effort required.

To illustrate the use of SPE in the software process, we will focus on the Unified Process. This process is iterative and incremental, and its features are typical of the process used for many object-oriented projects.

The Unified Process is divided into four phases: inception, elaboration, construction, and transition. One complete pass through these four phases constitutes a cycle that results in a product release. A product evolves over time by repeating the four phases in additional cycles.

The Unified Process is risk-driven. That is, the focus of each iteration is identified, prioritized, and performed based on risks. Risks are anything that might endanger the success of the project, including the use of new technologies, the ability of the architecture to accommodate changes or evolution, market factors, schedule, and others. Because the ability to meet performance objectives is a potentially significant risk, the Unified Process suggests dealing with this and other risks early in the process, when the decisions that you make are the most important and the most difficult to change later. The approach used by the Unified Process is to address important risks, such as performance, in the inception and elaboration phase, and to continue monitoring them during the construction phase. Risks are identified and managed using iterations.

Integrating SPE into the Unified Process is straightforward. When beginning a new project, you assess performance risk by evaluating the extent of use of new technology, the experience of developers, the complexity of the new software and operating environment, the scalability requirements for anticipated volumes of usage, and other factors. When planning another iteration, you evaluate the results of the previous iteration to determine if there is a performance risk. Feasibility models can quantify the achievability of performance goals. If there is a performance risk, you plan and execute the current iteration to reduce that risk.

Note: In cases where performance is critical and the risk is high, you might perform an iteration specifically to address performance concerns. This iteration might involve, for example, implementing or prototyping a critical component to provide measured values for model input, or to demonstrate that the component under consideration can, indeed, be constructed to meet its performance objective.

In the inception and elaboration phases, your knowledge of the details of the system's architecture and design are sketchy. As a result, during these phases, you will focus on best- and worst-case analyses using upper and lower bounds for the resources required in each processing step in the execution graph. Later, as your knowledge of the system's details improves, you can elaborate the model and refine your estimates.

2.4 Performance Solutions

Performance solutions are discussed in Part IV.

The quantitative techniques described in Section 2.2 form the core of the SPE process. SPE is more than models and measurements, however. Other aspects of SPE focus on creating software that has good performance characteristics, as well as on identifying and correcting problems when they arise. They include

- Applying *performance principles* to create architectures and designs with the appropriate performance characteristics for your application
- Applying *performance patterns* to solve common problems

- Identifying *performance antipatterns* (common performance problems) and refactoring them to improve performance
- Using *late life cycle techniques* for tough problems

An overview of each of these aspects of SPE follows.

2.4.1 Performance Principles

Performance Principles are presented in Chapter 9.

Constructing and solving performance models quantifies the performance of your software's architecture and design. After you have modeled a number of different designs, some of which have good performance characteristics and some of which don't, you will begin to develop a feel for what works and what doesn't. You will avoid those design strategies that have repeatedly produced poor performance and, consciously or unconsciously, incorporate those that consistently produce good performance into your standard "bag of tricks."

A set of general principles for creating responsive systems helps shorten that learning process. The performance principles help to identify design alternatives that are likely to meet performance objectives.

The nine performance principles presented in Chapter 9 generalize and abstract the knowledge that experienced performance engineers use in constructing software systems. They help to identify design alternatives that are likely to meet performance objectives.

2.4.2 Performance Patterns

Performance Patterns are presented in Chapter 10.

A pattern is a common solution to a problem that occurs in many different contexts [Gamma et al. 1995]. It provides a general solution that may be specialized for a given context. Performance patterns describe best practices for producing responsive, scalable software. Each performance pattern is a realization of one or more of the performance principles. Chapter 10 presents seven *performance patterns*. These are new patterns that specifically address performance and scalability.

2.4.3 Performance Antipatterns

Performance Antipatterns are presented in Chapter 11.

Antipatterns are conceptually similar to patterns in that they document recurring solutions to common design problems [Brown et al. 1998]. They are known as *anti*patterns because their use (or misuse) produces negative consequences. Antipatterns document common mistakes made during software development. They also document solutions, or *refactorings,* for these mistakes. Thus, antipatterns tell you what to avoid and how to fix problems when you find them.

Performance antipatterns document recurring performance problems and their solutions. They complement the performance patterns by documenting what *not* to do and how to fix a problem when you find one. This approach is particularly useful for performance because good performance is the absence of problems. By illustrating performance problems and their causes, performance antipatterns help build performance intuition.

2.4.4 Implementation Solutions

Implementation solutions are presented in Chapter 12.

If you have a poor architecture or design, it is unlikely that any amount of clever coding will enable you to achieve your performance objectives. This does not mean that later life cycle activities can be ignored, however. In fact, it is necessary to manage performance throughout the software development process to ensure that you will meet your objectives.

In an ideal world, once you have constructed and solved the SPE models to verify that your proposed architecture and design will meet your performance objectives, most of your work would be done. Your primary focus during the later phases of development would be to monitor the evolving software to confirm that the performance is as predicted, or to detect and correct any deviations that arise. For many systems, this may be all that is needed.

The real world is sometimes not so cooperative, however, so it is often necessary to carefully manage performance throughout the software development cycle. For example, some systems have a high risk of performance failure, or have other constraints (e.g., regulatory constraints for safety-critical systems) that require closer monitoring throughout implementation, along with careful attention to detailed design and coding activities.

You may find yourself with a system that has already been implemented and has performance problems. In this case, it is too late to apply the SPE techniques covered in other chapters from scratch. Instead, you must start with what you have and try to make it work.

Our systematic approach to tuning is presented in Chapter 12.

A systematic strategy for tuning poorly performing software that is based on quantitative data helps you to focus on the areas with the highest payoff, rather than expending effort on improvements that have a negligible overall effect.

Once you have identified the causes of the problems, you can correct them by applying the performance principles, using one of the performance patterns. Alternatively, if you have identified a performance antipattern, you can apply one of its refactorings. Chapter 12 discusses specific solutions for many common problems, as well as solutions specifically adapted for object-oriented software and the C++ and Java languages.

2.5 Summary

The SPE process focuses on the system's use cases and the scenarios that describe them. This focus allows you to identify the workloads that are most significant to the software's performance, and to focus your efforts where they will do the most good.

SPE begins early in the software development process to model the performance of the proposed architecture and high-level design. The models help to identify potential performance problems when they can be fixed quickly and economically.

Performance modeling begins with the software model. You identify the use cases that are critical from a performance perspective, select the key scenarios for these use cases, and establish performance objectives for each scenario. To construct the software model, you translate the sequence diagram representing a key scenario to an execution graph. This establishes the processing flow for the model. Then, you add software and computer resource requirements and solve the model.

If the software model solution indicates that there are no performance problems, you can proceed to construct and solve the system model to see if adding the effects of contention reveals any problems. If the software model indicates that there are problems, you should deal with these before going any further. If there are feasible, cost-effective alternatives, you can model these to see if they meet the performance goals. If there are no feasible, cost-effective alternatives, you will need to modify your performance objectives, or perhaps reconsider the viability of the project.

To be effective, the SPE steps described in this chapter should be an integral part of the way in which you approach software development. SPE can easily be incorporated into your software process by defining the milestones and deliverables that are appropriate to your organization, the project, and the level of SPE effort required. This chapter presented an overview of how SPE can be integrated into the Unified Software Process.

The quantitative techniques described in Section 2.2 form the core of the SPE process. SPE is more than models and measurements, however. Other aspects of SPE focus on creating software that has good performance characteristics, as well as on identifying and correcting problems when they arise. They include

- Applying *performance principles* to create architectures and designs with the appropriate performance characteristics for your application
- Applying *performance patterns* to solve common problems
- Identifying *performance antipatterns* (common performance problems) and refactoring them to improve performance
- Using *late life cycle techniques* to ensure that the implementation meets performance objectives

By applying these techniques, you will be able to cost-effectively build performance into your software and avoid the kinds of performance failures described in Chapter 1.