10

Software Design

Now that you have mastered the mechanics of developing quality products on predictable schedules, it is time to address more advanced topics. The first of these is design. Software design is important because good design is the key to scalability, and scalability is the key to producing cost-effective and high-quality systems. To appreciate the impact of scalability on software development, consider building a boat. A competent amateur could probably build a 10-foot rowboat in his or her garage. However, building a 1,000-foot ship would be a totally different matter. Scaling up a boatbuilding job by 100 times changes almost every aspect of the job. With software, however, we attempt to build million-LOC systems with the same design methods we used for 10,000-LOC programs. And for these 10,000-LOC programs, we often even use the same methods we used for 100-LOC programs.

If our small-scale methods were scalable, large-system development would not be such a serious problem. To be scalable, our design methods must start with a solid foundation and they must produce precise, complete, and correct design artifacts. We all make mistakes, so our development process must include thorough design reviews and analyses, and it must find and fix as many of the design defects as possible. The top priority must be design quality, not development speed. If our small program parts do not all have flawless designs, our large system projects will be troubled and possible even fatally flawed.

Because the design methods that work best for you depend on the kind of work you do as well as on your skill and experience, the PSP does not specify any particular design method. It concentrates instead on producing a precise design representation. This chapter discusses the need for good design practices. It also covers the concepts and strategies to consider in producing designs, as well as some common ways to address scalability problems when developing large software systems. Chapter 11 describes the design templates the PSP uses to ensure design completeness and precision, and Chapter 12 addresses design analysis and design verification.

10.1 What Is Design?

Physicists and engineers make approximations to simplify their work. These approximations are based on known physical laws and verified engineering principles. The software engineer has no Kirchhoff's law or Ohm's law and no grand concepts like Newtonian mechanics or the theory of relativity. We lack basic structural guidelines like strengths of materials or coefficients of friction. We can, of course, defer rarely used functions or temporarily ignore error and recovery procedures, but we cannot separate software procedures by their likely impact on the system's performance. With hardware, you can often determine that the performance of some elements will have a 10% impact on the system, whereas others will have less than a 1% impact. In the software business, minor details often make the difference between a convenient and functioning product and an unusable one. Because of the very nature of software, we cannot generally reduce the complexity of our products by making approximations.

The Power of Abstractions

We do, however, have some powerful tools at our disposal. We can create abstractions and arbitrarily combine them into larger abstractions. Even when we know nothing about the internal structure of these abstractions, we can name and use them as long as we know all of their important external specifications. Hence, we have the opportunity to essentially write our own rules. If we conform to the capacities and capabilities of the systems we support, we can create whatever logical structures we choose.

This freedom, however, has its price. We do not face the physical constraints of physicists and hardware engineers, but we do face intellectual ones. As our products become more complex, we are often unable to grasp all of their critical details. Although we are not constrained by the physical laws of nature, we do

need consistent rules and concepts. Because of human fallibility, however, our rules and concepts are often flawed, incomplete, or inconsistent. That is why design and quality management are such important issues for software engineering.

The Problem with Abstractions

Because the fundamental software development problem is scale, it is worthwhile to consider the various ways to address scalability. The most common approach is to subdivide the overall system into parts. However, depending on how you do this, you might still not have usefully partitioned the work. Suppose, for example, you defined a family of general-purpose, reusable parts that could implement an entire system. Although this would likely be helpful, the degree of help would depend on the size and nature of the parts. If the system were to have 1,000,000 LOC and you created 500 part types that each averaged 10 LOC, then the system design task would have been reduced to designing a system of 100,000 parts. Although this would seem to be a big help, it would probably not solve the system design problem. In fact it could even make it worse. The designers would now have to learn and become fluent with each of these 500 part types. They would have to both design and develop the parts and then design the system to effectively use them. Essentially, you would have designed a new programming language, rather than addressing the system scalability problem. Designers have limitations, and they cannot effectively use large numbers of unfamiliar parts. Merely devising a richer language may help, but doing this does not address the essential problem of scalability.

Object-oriented designers can encounter this same part-multiplicity problem unless they are careful. Once they define a comprehensive family of elemental system classes, they might then expect to just put these classes together to form the system. Although having such classes could help, part definition does not replace the need for system design. To have useful scalability, you must not only meet the requirements of physical scalability, but also capture significant system function in the parts.

Solving the Abstraction Problem

Another way to solve this abstraction problem is to follow the divide-and-conquer rule. That is, divide the overall system into ten or more smaller parts that, we hope, can each be viewed as a coherent subsystem or component. If we do this properly, we will then have reduced the overall problem by a factor of ten or more. Doing this properly, however, is a challenging task, and it is called design.

A good software design transforms an often ill-defined requirement into a precise and implementable design specification. For any but the simplest programs, it is nearly impossible to produce a high-quality implementation from a

poor-quality design. Design quality has two parts: the quality of the design concept and the quality of the design representation. Although you might view representation as the less important of the two, in many ways it is more important. Designs that are badly represented will almost always be poorly implemented, almost regardless of the quality of the design concept. A poor representation can also make the design so hard to understand that you won't recognize its conceptual problems until you implement it or even later.

10.2 Why Design?

One of the reasons why software quality and security problems are so common is that few software developers take the time to produce thorough designs before they start coding. Although there are many reasons for this, the principal one is that when they spend much time on design, it takes them longer to develop programs. As you can see from Figure 10.1, the fastest programmers spent the least amount of time on design work.

If all you had to do was write small programs, and if these programs didn't have to work flawlessly in some large system, then design practices might not

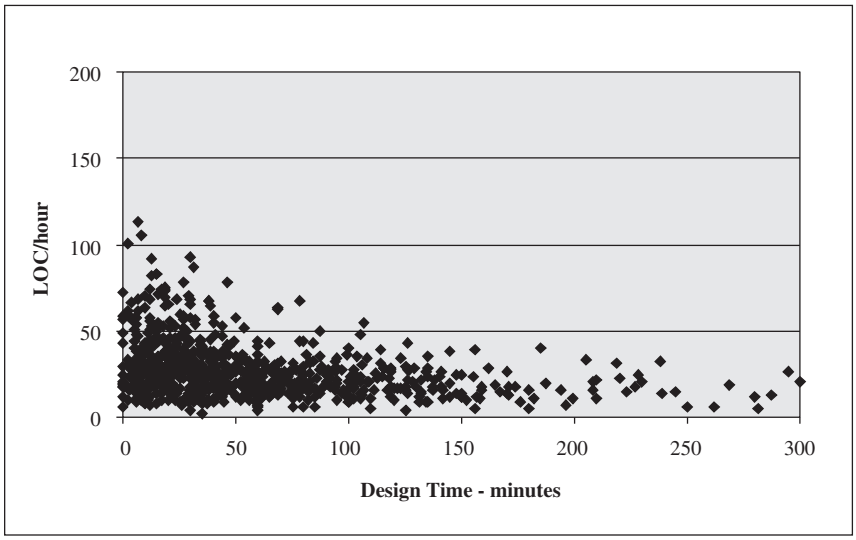

FIGURE 10.1 DESIGN TIME VERSUS PRODUCTIVITY (810 DEVELOPERS, PSP PROGRAM 6)

seem important to you. However, the need for safe and secure software systems is increasing daily, and safety and security cannot be assured without a sound design and disciplined quality management. Furthermore, if you don't practice sound design methods with module-size programs like the PSP exercises, you won't have the skill and experience to do a quality job when you really must produce a design.

The PSP strategy is to write every program as if its quality were critical. To do this, you must measure your work, produce thorough designs, and follow the quality-management practices described in Chapters 8 and 9. This is important because design quality is a trade-off. Producing a clear, complete, and defect-free design will slow you down as an individual developer. However, if you and your teammates all produce clear, complete, and defect-free designs, it will accelerate the team's work.

10.3 The Design Process

Software design is a creative process that cannot be reduced to a routine procedure. This process, however, need not be totally unstructured. Design work generally starts by reviewing the product's purpose, gathering relevant data, producing an overview design, and then filling in the details. These steps are not isolated sequential tasks, however, but are often highly iterative parallel activities. Design involves discovery and invention, and it frequently requires intuitive leaps from one abstraction level to another.

For complex designs, good designers follow a dynamic process. They work at a conceptual level for a period and then delve deeply into a particular issue. When they are unsure how to proceed, they write some code. Curtis, a well-known expert on software development practices, found that this behavior is normal and that it produces the insights needed for high-quality designs (Curtis et al. 1988). This dynamism was a trait of all the best designers he studied.

At the outset, no one will really understand the requirements, the design, or the implementation. As each area is refined, it sheds further light on the others. This is particularly true once development reaches the point where users can try early product versions. Each such exposure leads to new knowledge, new ideas, and more changes. The most sensitive development decision concerns the precise time to truncate this learning process to produce a suitable product on a reasonable schedule and at an affordable cost.

Requirements Uncertainty

For any but the simplest programs, the design process can be quite complex. Figure 10.2 shows some of the more important elements of this process and one way

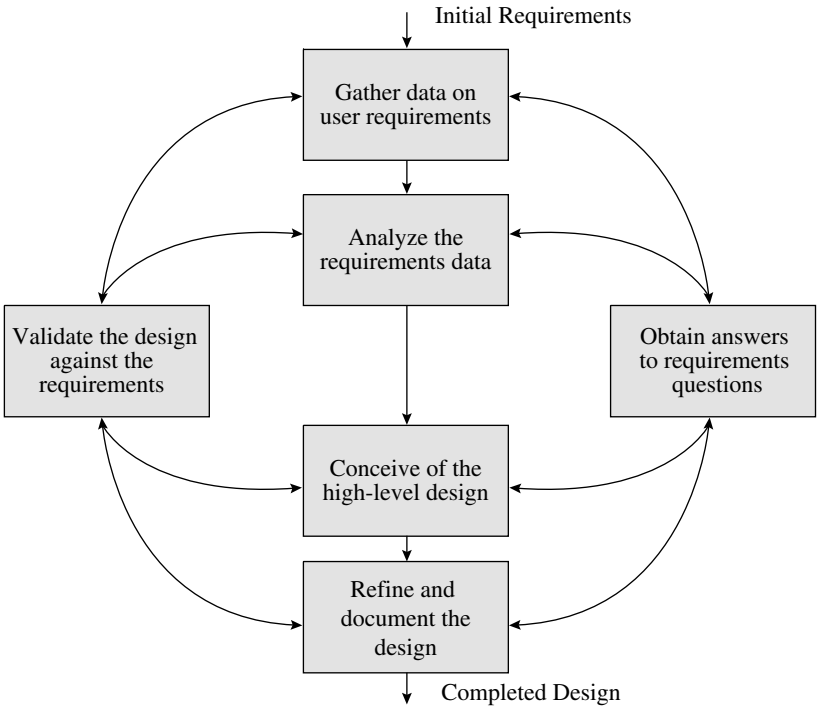

FIGURE 10.2 THE DESIGN FRAMEWORK

to relate them. In the simplest case, assuming the requirements information is current and properly communicated, this process can proceed directly from top to bottom. Of course, this assumes there are no false starts or changes. However, in the real world, misunderstandings, errors, and omissions are common, and properly handling the resulting changes takes many feedback loops and iterative cycles.

This creative design process is complicated by the generally poor state of most requirements descriptions. This is normal for new software systems because of what I call the **requirements uncertainty principle**: the requirements will not be completely known until after the users have used the finished product. The users' jobs will then have changed, as will their needs and requirements. Therefore, the true role of design is to create a workable solution to an ill-defined problem. Although there is no procedural way to address this task, it is important that the designers treat design as a discovery process and be alert for any new information that might impact their initial design approach.

Projects rarely get into trouble because of the occasional massive requirements change. Such large changes are typically recognized and properly reviewed. The

most dangerous ones are the many seemingly trivial enhancements that can nibble you to death. If you do not have the knowledge, skill, and discipline to analyze the impact of every small change, the scope of the job can quickly double or triple. The PSP process will help you to control this change spiral. It enables you to make accurate plans and to estimate the probable impact of even the smallest change. These plans provide you, your teammates, and your managers with a rational basis for deciding when to freeze and build and when to make further changes.

Design as a Learning Process

Once you have a reasonably clear understanding of the high-level requirements, you will probably know enough to produce a high-level design. At this point, the requirements data-gathering process changes. You now seek information that will either validate this initial design approach or at least define its scope and limits. What are the performance specifications and size limits? Are there special peak demands or unusual response needs? What must the users know, and what are their likely skills? What is the best way to handle the security, privacy, and safety issues?

To keep from getting lost in a mass of detail, "walk around" the problem and look at it from every angle you can imagine. Think like a user and visualize the likely scenarios. Then walk through these scenarios to ensure that the high-level design handles them naturally and smoothly. If you identify too many exception conditions, this design may not be a natural fit to the problem. Once you have corrected and validated the high-level design, it provides the framework for cataloguing the more detailed knowledge you gain throughout the development process.

Even though the design process is dynamic, it is still helpful to think of it in process terms. This is not to constrain design behavior but to help define the design products and provide guidance in producing those products. A structured design process also helps manage the dynamics of the work. While mentally jumping from one artifact to another and simultaneously considering several design levels, a process framework helps to maintain control of what you are doing. The process also helps you track design status.

Prototyping

Although complex problems often have complex solutions, that is not always the case. The principal exception is when you can identify elements of the problem that have been solved before. In this case, you may be able to reuse part of that previous solution. Similarly, simple problems do not always have simple solutions. Often, the simplest requirements cause the most problems. Application functions that are very simple for people to conceptualize are often extraordinarily hard to design and implement.

This is why, as we design any but the simplest products, we should regularly consider the following five points:

1. Do I need to refine this design further?
2. If I do, do I need to do it now?
3. If I need to make the refinement now, do I know enough to do so?
4. If I don't need to do the refinement now, am I confident I will be able to do so when I need to?
5. If I don't know enough to refine the design, I must address that problem now.

If you don't know enough to refine your design, consider developing a prototype of the potentially troublesome functions before proceeding. If your design is based on functions that you might not be able to implement, resolve that uncertainty as soon as you can. If you don't, there is a good chance that you will waste much of your subsequent design work. That is a principal reason for developing prototypes. Occasionally, you will not be able to even specify a function until you have designed and possibly built and tested it. In these cases, develop prototypes for these functions before completing the design.

Prototyping is a powerful tool that can be used in many ways. You may want to experiment with reusable program elements to verify that they work the way you want them to work. You might need to understand a program's real-time response capability, minimize memory usage, or demonstrate some function to a potential user. One useful quality practice is to build a prototype every time you use a new or unfamiliar library function. These small throwaway programs are intended only to answer questions; they will not be part of the finished product.

Before developing a prototype, specify its purpose and define the questions it is to answer. In the PSP, prototype work is part of design. If you later want to use the prototype code in the product, then all of the design and coding work required to bring the prototype design and code up to product standard is tracked as a normal part of development. If the prototype code is not to be part of the product, there is no need to measure its size or track the defects you found while developing it. Of course, the prototype development time is recorded as design time.

10.4 Design Levels

Unfortunately, there is no standard nomenclature for levels of design. In this book, I discuss the following five design levels:

1. **The conceptual design:** This is not really a design level but more of a planning concept to help you estimate a product's size.

2. **System-level design (SLD):** This is the overall design for the highest-level view of the entire hardware-software product. One way to define the system level is that level above wherever you work. For example, if you worked on automobile engines or brakes, the system level would be the car, but if you worked on pistons or carburetors, the system level would be the engine.
3. **High-level design (HLD):** This design level spans the range from immediately below the system-level design to immediately above the detailed-level design. The role of HLD is to divide whatever product level you are currently addressing into smaller parts.
4. **Detailed-level design (DLD):** The DLD level starts with the lowest-level parts specified by the HLD and defines how to build them.
5. **Implementation-level design (ILD):** The implementation takes the DLD and refines it into a form from which the product can be automatically constructed.

It helps to think of the design process as an inverted pyramid (see Figure 10.3). Each level provides a foundation for the next. During high-level design, you deal with structural, performance, security, safety, and functional decisions. If you do not make and document all of the necessary high-level design decisions in high-level design, you will have to reconstruct them again during detailed design. If you don't

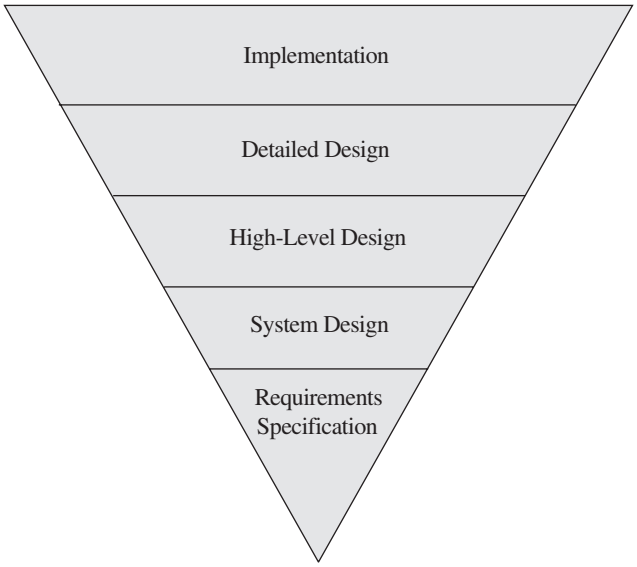

FIGURE 10.3 THE DESIGN PYRAMID

make and document all of the detailed-design decisions in detailed design, you will have to reconstruct them during implementation. Even when you are the implementer, reconstructing the reasons for and content of all of your mentally completed but undocumented designs is both time-consuming and error-prone. During implementation, you think at an implementation level. Without considerable effort and a complete change of mind-set, you will have trouble reconstructing the relevant high-level and detailed-level design information you need to produce a quality implementation. This is when design errors are most likely.

Design level is a concern during every process phase. For example, when you resolve requirements issues during high-level design, or high-level design issues during detailed design, you will also likely make mistakes. The typical requirements-specification-design process for large projects is shown in Figure 10.4.

Briefly, the design steps called for in this figure are as follows:

1. Define the need (requirements definition).
2. Define the solution (system specification).
3. Conceptualize the solution (system high-level design, or HLD).
4. Subdivide the work (product specifications).
5. Define the product design (product HLD).
6. Subdivide the products into components (component specifications).
7. Define the component design (component HLD).
8. Subdivide the components into modules (module specifications).
9. Detail the solution (module detailed design, or DLD).
10. Implement the solution (module implementation and test).

The specifications and high-level designs are progressively refined until you reach the detailed-design and implementation levels. Only then are you ready for implementation. Again, however, this is not an orderly top-down process. Depending on the nature of the problems, the first step in high-level design could be to implement some critical but poorly understood module.

Requirements Definition

Even when a systems engineering or requirements group is charged with documenting the requirements, what they produce will rarely be adequate for your needs. To produce a competent design, you must have accurate and complete requirements and you must understand them. You could argue that you should have complete requirements before starting the work, but that is rarely possible, for several reasons:

- Requirements specification is a specialized skill, and one for which few people have the requisite knowledge and experience.

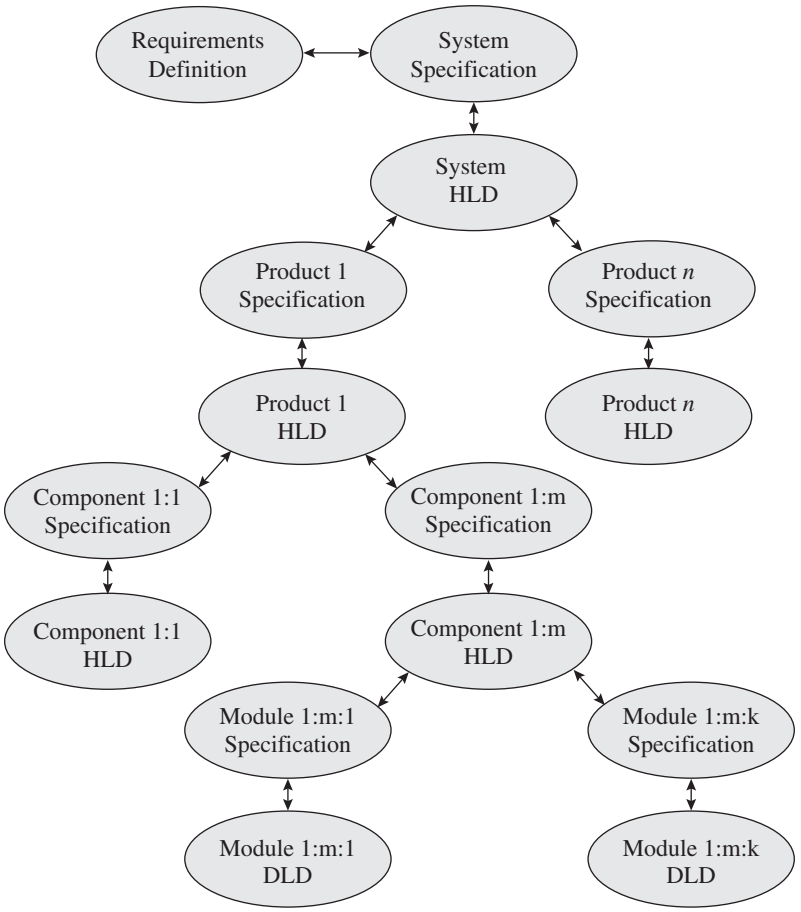

FIGURE 10.4 THE REQUIREMENTS-SPECIFICATION-DESIGN CYCLE

- Requirements change. As your design progresses, you will ask questions that cause the system engineers and/or customers to think more deeply about their needs. This will generate new ideas. Even if you do not stimulate this process with probing questions, the longer you take to produce the design, the more likely they are to think of new or changed needs.
- The solution you develop will often change the problem. Because many computer applications concern new services, using the product changes how people work, which probably changes their needs. These changes then impact the requirements.

The third reason is the requirements-uncertainty principle and it has major implications for the development process. Although you can often hold the customers (or system engineers) responsible for the requirements they produce, your principal concern is to produce a quality design, not to assign blame for requirements oversights and errors. Therefore, you must learn as much as you can about the requirements. Once you have a clear and complete requirements definition, you can proceed with confidence to specify the design. If the answers you get are confused or incomplete, focus on helping the users (or system engineers) produce a complete and precise requirements statement. Do not produce the requirements statement for them, but propose design solutions, postulate test scenarios, or build prototypes or initial product versions to help them better understand their own needs.

The Design Specification

The design specification should be a complete and precise statement of what the program must do. Although using spoken language can produce reasonably clear specifications, a truly precise specification must be written in a truly precise language such as Z (Jacky 1996). The requirements can be viewed as a problem statement without a solution. In fact, if the requirements are properly produced, they should not imply a design. The specifications, however, postulate the existence of a solution and state its constraints and invariants. For small products, the specifications can often be a simple extension of the requirements. However, for larger products, the specifications should be completely separate from the requirements.

As shown in Figure 10.4, the requirements definition and system specification are developed at the highest level. As development proceeds, you divide the system into major products, each of which must be specified and designed. These products, in turn, may be composed of several components that must also be specified and designed. Thus, the specification must be refined for each product level.

The system specification is inextricably linked with the requirements definition. It takes considerable effort to subdivide the system's functions into coherent products and components and to specify their relationships. If these design decisions are not precisely recorded when made, they must be re-created later. For example, this will be necessary whenever there are system-level questions on safety, security, or performance. Trying to document these decisions later is difficult, error-prone, and time-consuming. Once you have clarified some aspect of the requirements, promptly reduce it to a rigorous specification. Then address and specify the next system aspect. If this specification conflicts with previously completed specifications, make the needed changes as soon as possible. Any delay will likely either delay the design work or cause errors.

The more precise the specifications, the more likely you are to identify requirements mistakes, omissions, and inconsistencies. The precision of the specification is much like the resolving power of a lens. An imprecise specification

provides a vague and out-of-focus view of the system, whereas a precise specification brings requirements mistakes and uncertainties into clear focus. In resolving these issues, you will often have to retrace some of your earlier decisions, so it is helpful to include specification notes that explain why you made each key choice.

Although this may sound like a sequential process, it is not. Every specification question can force a requirements change and every design question can impact the specifications. These phases must be dynamic so specification issues can be resolved rapidly. With a clear and explicit specification, you can confidently proceed with the design.

High-Level Design (HLD)

In the high-level design phase, you create the overall design. This involves decisions about reusing existing components, defining classes, and specifying class relationships. The high-level design must consider two different systems: the application system and the reusable component library. Any potentially reusable functions should be designed to be part of the reusable component library, and the application-specific functions should be tailored to the unique needs of the product. Because reuse can save a great deal of time and effort, capitalize on the reuse library during every development phase, starting with HLD.

HLD is also where you make trade-offs between product content, development cost, and the schedule. An elegant design must balance development economics, application needs, and technology. By properly relating these three dimensions, you can balance the feasible with the desirable and the affordable. Although you may not produce a product that does everything the users want, you should aim to provide one that neatly and cleanly does what they need and can afford.

During HLD, many questions arise and many redesigns must be considered. You, the customers, and the system engineers must be in close and continuous contact. During the HLD and requirements work, these trade-offs are generally easy to make but are largely a matter of guesswork and judgment. However, some functions can be very expensive, and their inclusion could jeopardize the project. This is why you must give the users credible estimates for the cost and schedule impact of each change. Cost and schedule are important design considerations, and when users do not appreciate the business consequences of their requests, development disasters are most likely. By precisely describing the costs of a proposed change, you can help the users and system designers make logical and businesslike decisions about what they really need and can afford.

Detailed-Level Design (DLD)

The next design phase is detailed design. Here, you reduce the HLD to implementable form by detailing the functions, specifying the class state machines, and

producing the finished design. Some of the HLD concepts might be unimplementable, while others could be unsound. You could see reuse opportunities, have performance insights, and think of integration and system testing issues. Because these changes can impact the requirements and HLD, neither the requirements nor the HLD can be considered firm until you have completed the DLD.

Implementation-Level Design (ILD)

The final design phase is implementation. Although writing code is not usually considered design, it does involve design issues. Not only will low-level design choices remain after DLD, but as you address implementation issues, you will generate questions about the design. Some of these will require design changes, some may lead to design improvements, and some may even cause requirements changes. In a sense, then, the actual code-writing process is the first real test of the design.

10.5 Design and Development Strategies

Your design approach must be closely coupled to your development strategy. In essence, design by abstraction consists of successive disintegrations of the system into manageable pieces. Implementation consists of building and integrating these pieces into a coherent whole. If the design strategy is a poor fit to the problem, you can waste time constructing special test drivers and making multiple program modifications. A poor development strategy is also likely to produce a confused, hard-to-understand, and hard-to-modify design that has quality, usability, security, safety, or performance problems. Because the development strategy must reflect the product's unique needs, there is no best strategy for all systems.

With the PSP exercises, you can use a simple development strategy. However, on a TSP project, you and your team will define your own design and development strategy and process for each project. Before defining them, however, consider the design levels needed for that project and the proper way to address them.

Strategy Guidelines

In designing large programs, you will often be unable to specify all of the program's highest-level functions until you know more about some of the functions at a lower level. You might leave some higher-level specifications incomplete until you have done some lower-level design work. Starting at the top, you would

proceed until you hit an unknown abstraction. Then, if this abstraction were particularly difficult or critical, you might design it before proceeding. This abstraction in turn might call others that appear equally critical, and those could lead to others. You could find yourself delving deeply into the system before you complete the highest-level design. On occasion, you may even find it desirable to work from the bottom up or in some other order. Although this sequence would seem quite logical to you, it could look to an uninformed observer like a random series of disconnected partial design steps.

When following a dynamic development strategy, consider the following guidelines:

- Where practical, complete the highest-level designs first.
- Do not consider the design of a program or a major program element complete until the specifications for all of the abstractions it uses are complete.
- Do not consider the design of a program element complete until the specifications for all of the program elements that depend on this element are also complete.
- Log the assumptions you make so you can later check to ensure that they are valid. It is good engineering practice to maintain an issue tracking log to record outstanding issues, questions, or problems.
- Make a practice of writing design notes that explain the logic for your design choices. Make these notes either in the design documentation itself or in your personal engineering notebook.
- Resolve the uncertainties that prevent you from specifying an abstraction before you attempt to design it. It may be helpful to use prototype experiments to do this.
- Penetrate as many design levels as needed to resolve specification uncertainties.
- Defer resolving lower-level design uncertainties if there are no feasibility concerns and these lower-level designs do not impact other system elements.

The development strategy provides the framework for disintegrating and reintegrating the product. A well-founded strategy must consider the logical structure of the ultimate product. By producing a strategy that naturally fits the product's structure, you can usually simplify the design job and accelerate development.

Example Development Strategies

Suppose you were developing a product with a central control element and a family of application classes. Suppose further that each class used additional classes, which in turn used others. You could visualize all of these classes as a tree, with

the control element at the top calling those classes immediately below, and so on. Classes that are used in several places would appear in multiple places on the tree.

After dividing this total tree into PSP-implementable elements, you must decide on the order for developing and integrating these elements. One approach would start with the top element and work toward the bottom of the tree. Another approach would start at the bottom and work up. Obviously, a third approach would be to start somewhere in the middle and work in both directions. You could also use hybrid strategies, such as developing multiple top-to-bottom threads or slices of the entire system.

These approaches each have advantages and disadvantages. A top-down strategy ensures that each cycle has a clear operational context and well-defined requirements. The reviews and tests can then realistically address operational issues. The principal problem with the top-down approach is that the lower-level abstractions must be stubbed or bypassed. Now, because some of these functions may be incompletely defined, the higher-level functions may misuse them.

Starting from the bottom and working up has the advantage of working with a solid foundation. You first develop those leaf elements that use no undeveloped abstractions. As you add layers, you are building on proven designs and are less likely to run into problems with overlooked details. The principal disadvantage of the bottom-up strategy is that you are implementing component specifications that may not do precisely what the system requires. You may, for example, develop a function and then later discover that it was improperly specified. Another potential problem is the lack of test drivers. With the bottom-up strategy, the lower-level abstractions will not typically form a coherent system. Such a strategy will not provide a natural test environment for most of the abstractions.

The third approach—starting in the middle—could be advantageous when a critical function determines the success of the entire program. An initial focus on that function could be appropriate if it is unusually complex or if you suspect it presents special problems.

The example strategies described in the following sections are the **progressive**, **functional-enhancement**, **fast-path**, and **dummy** strategies.

The Progressive Strategy

The progressive strategy is a natural way to develop systems that consist of sequentially executing functions. You first develop and test the operations that are performed first, then add and test the next functions, then the next, and so forth. If there is some central control structure, you can develop it in gradually widening vertical slices. The classes that support each slice are then implemented from top to bottom in a complete development iteration or cycle. As each cycle is designed, implemented, and tested, the pipeline is gradually lengthened to include additional functions.

This strategy has the principal advantage of being easy to define and to implement. It needs test scaffolding only for special cases, and there is little to rebuild

or redesign between cycles. One disadvantage is that there may be no easy way to exhaustively test the behavior of each slice. You thus may need special scaffolding to test functions that are not directly visible to users. However, this would likely be a problem for such functions with almost any development strategy.

The Functional-Enhancement Strategy

The functional-enhancement strategy defines an initial stripped-down system version, or kernel, and adds enhancements. This is the common strategy for follow-on enhancements to large systems. A base system kernel is initially defined that need not be functionally useful. The only requirement is that it provide an operational context for subsequent enhancements. This kernel should be as small as possible and it must be carefully reviewed and tested. The added functions are then incorporated in small steps.

The functional-enhancement strategy builds a working system at the earliest point. Because its initial focus is getting a working system, it often leads to earlier identification of overall system problems. It also provides an early base for safety, security, and performance testing and for preliminary user trials. This strategy is also the natural backup case for most large-system development projects. When developers attempt to deliver too much function with the first release, they usually end up paring functions and shipping a minimum initial system. The principal debates with this strategy concern the functions to ship with each release. The principal problem with this strategy is that the first step is often quite large.

The Fast-Path Enhancement Strategy

The fast-path strategy is much like functional enhancement except that the initial system kernel is designed to demonstrate fast-path performance. The critical timing issues are identified and the basic real-time control cycle is implemented. The focus is on performance, so this initial fast-path kernel should contain only the minimum logic needed to control the performance-sensitive functions.

The principal advantage of fast-path enhancement is that it exposes timing problems at the earliest point. Although the first system kernel may provide no recognizable external functions, performance can be measured and overall system performance estimated. After the kernel's fast-path performance is demonstrated, functional enhancements can be added incrementally. System performance can be measured with each enhancement and steps taken to ensure that it does not deteriorate. You will know early on if kernel performance does not meet the performance requirements, and have the maximum amount of time to fix the problems. For time-critical systems or for the initial release of an operating system, this can be a very attractive development strategy.

Its principal disadvantage is also the size of the first step. Although this kernel could be smaller than for the functional-enhancement strategy, it could still be too large to conveniently handle with a single development team.

The Dummy Strategy

The dummy strategy follows a top-down sequence. It starts with high-level system functions and dummies the rest with stubs that provide predefined returns. After the highest-level functions are tested, succeeding cycles gradually replace the stubs with implemented functions. This is an appropriate strategy for layered systems or for the kernel portion of fast-path or functional-enhancement systems. This strategy has the advantage of building the system in small steps and it provides considerable flexibility in ordering the functional implementation. In building the kernel for a fast-path development strategy, for example, the initial control loop would probably need multiple stubbed dummy functions. Each of these functions could then be more or less independently developed and incorporated in the growing system.

The principal disadvantage of the dummy strategy is that useful system behavior is often not visible until late in development. This could make it difficult to do early testing of critical system functions.

Selecting a Development Strategy

One potential problem with any strategy is that some critical function may not be developed until the later development cycles. This would delay exposing key project risks and possibly result in initially testing some complex system function in the full system. If the system has many defects, this could greatly delay and complicate testing. It is generally wise to test the most difficult components with the smallest amount of untested code. The defects are then more easily identified and fixed.

There is no one best strategy. You must examine the system structure, assess the principal development risks, and select a strategy to fit each situation. The goal is to have the strategy naturally fit the system structure and to expose the principal risks as quickly as possible.

10.6 Design Quality

The software design should contain a sufficiently complete, accurate, and precise solution to a problem to ensure its quality implementation. Because a design has many potential users, the following sections discuss these issues from the perspective of the design's users.

Design Precision

The lack of a precise design is the source of many implementation errors. By examining your PSP data, you will likely find that many of your design defects were caused by simple errors. The reasons will vary, but a common cause is the lack of a

properly specified design. In some cases, you may have completed the design but not documented it. At other times, you may not even have produced a complete design. If you don't have a precise specification for what the design must contain, your decisions about where to stop will generally be ad hoc and inconsistent. Often, in fact, you will think, "Anyone would know how to do this," and not bother to document it. However, without a precisely documented design, you or any other implementer must finish the design during implementation, which is highly error-prone.

Design Completeness

You can waste time overspecifying the design, but an underspecified design can be expensive and error-prone. The complete specification of a software design requires a great deal of information: defining classes and their relationships, identifying class interactions, defining required data, specifying system states and transitions, and specifying inputs and outputs. The complete and unambiguous definition of all of this material generally requires significantly more documentation than the source listing of the finished program. The source code must include all of the design decisions but in very terse form. A program's uncommented source code is very dense, has little redundancy, and precisely defines the product's behavior. With the possible exception of the object code, the source code is the most compact way to completely define a program's design. Unfortunately, unless the source code is extensively commented, it is almost unintelligible to anyone but the designers and implementers, and even they can have trouble deciphering it. The key question is, how much detail must the designers provide to the implementers and other design users?

A fully defined design is expensive, and much of its content is not needed by experienced implementers. In addition, particularly if you are both the designer and the implementer, overspecifying the design will waste time. To be most efficient, make an explicit design-content trade-off. This requires a specification of design completeness. Such a specification will help you to produce a quality design and will guide your design reviews. As long as you meet the design exit criteria, you can use whatever design methods and notations you prefer.

The Users' Needs

To properly specify the products of the design phase, consider the needs of the people who will use the design. The principal users are the implementers, the design and code reviewers, the documenters, the test developers, the testers, the maintainers, and the enhancers. With the TSP, you are the principal user of the design, but not the only one. However, for the PSP exercises in this book, you are the only user of your designs.

On a TSP project, everyone you work with will need access to all of your design information. Although the design materials must be generally available, they must also be controlled. Each design product must have an author and an owner. The owner should be the only person who can change the design. The following sections describe the various categories of owners and the design products they should control.

System or Product Management

The items that should be owned, tracked, and controlled by system or product management are as follows:

- **Issue tracking log:** A running list of open issues and questions that must be resolved as the design progresses. These are like defect reports: anyone can submit one, resolution responsibility must be centrally assigned, the open issues should be tracked, and the resolution must be recorded and communicated.
- **Specification of any implementation constraints:** Coding specifications and configuration management standards and procedures.
- **A precise description of the program's intended function.**
- **Application notes:** Descriptions of how the product or system is to be used. These include a statement of how the user will use the products, including explanations of special conditions, constraints, or options.
- **System-level user scenarios:** Example usage scenarios.
- **System constraints:** Timing, size, packaging, error-handling, security, and hardware and system interface specifications.

System Engineers

Depending on the particular way the design was handled, the following items could be produced and owned at either the system or the product-design level:

- **A description of all relevant data and files:** The precise specification of the data to be provided and their structure
- **A description of system messages:** In a communication system, the specification of all message types, including formats, routings, and any relevant performance, security, or integrity specifications
- **Any special error checks or conditions**
- **Reasons why the system design choices were made**

Software Designers

The following basic design information is the sole responsibility of the software designers:

- **A picture of where the program fits into the system**
- **A logical picture of the program itself**
- **A list of all related classes, components, or parts:** Scope, class structure, functions provided, where initialized, where terminated
- **A list of all external variables:** Includes, scope, limits, constraints, where declared, where initialized
- **A precise description of all external calls and reference**s
- **A clear statement of the program's logic (pseudocode)**

A complete and comprehensive set of design products that meet these needs would be large, unwieldy, and confusing. They could also be a major source of error unless there were some automated way to ensure their self-consistency. Any reasonably large system includes many design changes. Thus, with inadequate change control, design changes will not be completely reflected in all of the design records. These design records will then become inconsistent, and anyone who uses them could be misled. This change control problem is vastly complicated whenever you record the same information in multiple places. It is therefore essential to specify the absolute minimum of required design information and to ensure that it is always kept up to date.

Although these design materials must be produced in some form, their content and format will vary considerably depending on the size of the system, the application domain, and the product standards. For the PSP, however, some design elements can be more or less standardized—those listed as the software designers' sole responsibility. They form the base for the PSP design process. The PSP thus concentrates on defining standards for this basic design subset. These standards and a defined way to meet them are covered in Chapter 11.

10.7 Summary

The size of software projects has increased rapidly for over 30 years. The products you develop in the future are thus likely to be much larger than today's. Because a process that is optimum for a small program will not likely be optimum for one that is 5, 100, or even 10,000 times larger, you will almost certainly have to change your development process and probably keep changing it for as long as you do software development work.

Many of software development's historic problems have stemmed from a misplaced expectation that development should start with firm and complete requirements. History demonstrates, however, that for a new software system, the

requirements will not be completely known until after you have a working product, and possibly not even then. In defining the design process, you must therefore recognize that the requirements will likely change—and take steps to identify and resolve requirements uncertainties as early in development as possible.

The design phase of large-scale software systems typically starts with a high-level system design effort that subdivides the product into components. These components are separately developed and then the system is integrated. Because these components are much smaller than the total system, they can presumably be developed with smaller-scale methods.

A well-founded development strategy is built on the natural structure of the planned product. Although there are many possible development strategies, they are all different forms of divide-and-conquer: disintegrate the system into multiple parts, develop the parts, and then integrate the parts back into the finished system. The strategy concerns the particular way to do this integration and disintegration.

There are two different aspects to software design: how to produce the design and how to represent that design once it is produced. Although the PSP does not specify a particular design method, it does define the exit criteria for the design phase. These exit criteria are based on a standard that defines the contents of a completed design.

References

Curtis, B., H. Krasner, and N. Iscoe. "A Field Study of the Software Design Process for Large Systems." *Communications of the ACM* 31, no. 11 (November 1988).

Jacky, J. *The Way of Z: Practical Programming with Formal Methods.* Cambridge, UK: Cambridge University Press, 1996.