Chapter 1

Software Architecture Basics

In This Chapter

- Understanding the basics of software architecture
- Finding the problem
- Identifying requirements
- Considering your software development style

COPYRIGHTED MATERIAL

The term *software architecture* means different things to different people. To the developer, it means the structure of the system being built. To the framework developer, it's the shape of the system that is created with the framework. To the tester, it's the shape of what needs to be tested. For all concerned, it's the high-level structure of the solution to a problem that the customer or client wants solved.

In this chapter, I explain the basics of software architecture — what it is and how you get started. Knowing the problem that you're solving and the important requirements of the system are also very important, and I help you get going with these tasks in this chapter. In Chapter 4, I explain how software patterns fit into the picture.

Understanding Software Architecture

Every system has an *architecture* — some high-level structure that underlies the whole system. *Software architecture* is how the pieces fit together to build the solution to some business or technical need that your customer or client wants solved. The architecture has a purpose.

The decisions made during the creation of the architecture are truly fundamental to the system because they set the stage for all the other decisions that will come later.

Some systems' architectures are best described as a Big Ball of Mud (see Chapter 2). These systems are hard to build and hard to maintain, and they may not meet the customer's needs. Tackling the development of a software system with good software architecture will lead to a more successful result.

To an unsophisticated customer or client, *software architecture* is a meaningless term, so don't get hung up trying to explain how wonderful your architecture is. The customer wants the finished product that solves the problem at hand, not a description of the software that you'll build to solve it. (For more information on explaining software architecture to others, see Chapter 3.)

Components of software architecture

The software architecture provides the high-level view of the system you're building and must cover the following aspects:

- **Goals and philosophy of the system:** The architecture explains the goals and describes the purpose of the system, as well as who uses it and what problem it solves.
- **Architectural assumptions and dependencies:** The architecture explains the assumption made about the environment and about the system itself. The architecture also explains any dependencies on other systems or on the builders of the system.
- **Architecturally significant requirements:** The architecture points to the most significant requirements that shaped it.
- **Packaging instructions for subsystems and components:** The architecture explains how the parts of the system are deployed on computing platforms and how the parts must be combined for proper functioning. The subsystems and components are the building blocks of the architecture.
- **Critical subsystems and layers:** The architecture explains the different views and parts of the system and how they relate. It also explains the most critical subsystems in detail.
- **References to architecturally significant design elements:** The architecture describes the most prominent and significant parts of the design.
- **Critical system interfaces:** The architecture describes the interfaces of the system, with special attention to the interfaces that are critical to meet the system's requirements.
- **Key scenarios that describe critical behavior of the system:** The architecture explains the most important scenarios that illustrate and explain how the system will be used.

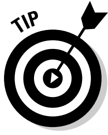
Architecture document

All the components in the preceding section go into an architecture document, which contains the information needed to interpret the architecture. The document includes assumptions, key decisions that shaped the architecture, how the parts of the architecture work together, and how the system will be packaged. I tell you more about the architecture document in Chapter 3.

Architecture models (views)

The software architecture has several audiences, including fellow architects, programmers, configuration managers, testers, and customers. All are interested in different information, and all look for different things within the architecture. To make your architecture useful to all these audiences, divide the architectural description into four different models or views:

- **Logical:** Maps the system onto classes and components. The logical view is directly related to the functional requirements, which I discuss later in this chapter. The logical view focuses on the parts of the system that provide the functionality and that the users of the system will see when they interact with it.
- **Process:** Explains how the parts of the architecture work together and how the parts stay synchronized. It also explains how the system is mapped onto the units of computing, like processes and threads. *Processes* are groups of tasks that together make something that can execute and perform the desired functions. The process view brings in some nonfunctional requirements (see "Defining nonfunctional requirements," later in this chapter), which aren't directly related to visible functions.
- **Physical:** Explains how the software that implements the system is mapped onto the computing platforms. The various components of the system, networks, processes, tasks, and objects are mapped onto the tangible parts of the system in the physical view. This view contains information related to the system's nonfunctional requirements (discussed later in this chapter), such as availability, performance, and scalability.
- **Development:** Explains how the software will be managed during development. The software will be written in small pieces that individuals or small teams can work on together. The development view highlights these pieces and shows how they are intertwined and interdependent. The development view reflects any limitations on the organization of the software based on limitations in the programming language, development environment, or development organization.



I tell you about diagram styles to use for each of these models in Chapter 3.

These four models of the system are usually supplemented by one additional view that describes common scenarios, tying the other views together by showing how elements within all of them work together. (Use cases, discussed later in this chapter, describe the scenarios.) This additional view is frequently called the *4 + 1 model.* Figure 1-1 shows how the parts relate. A good architecture balances all these views so that no view contains much more detail than any other.

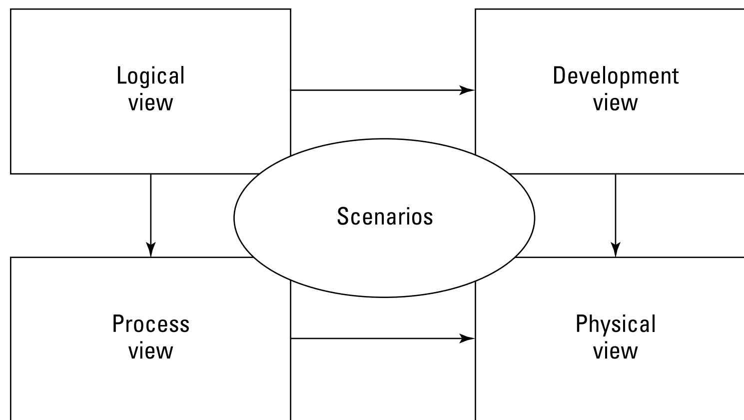


Figure 1-1: The 4 + 1 model of an architecture.

Software development methods and processes

Software development can be done in many ways. These different ways are called *methods* or *processes.* Here are a few examples:

- **Waterfall method:** In the *waterfall method,* the different phases of system development activities follow each other sequentially. The artifacts produced during development are considered to be flowing downstream and going over a waterfall between the analysis, requirements-gathering, development, and testing phases of development. Artifacts always move forward, or downstream, without repeating a phase more than once.
- **Unified Process:** The *Unified Process* is a popular process in which the various activities — such as requirements generation, development, and testing — overlap. Instead of being associated with particular work products and the tasks that create them, the phases in the Unified Process follow the life of the product, from inception to elaboration to construction and finally to transition. Within each of these phases, the activities are iterated, always focusing on the most critical aspects.

- **Agile methods:** Agile development methods are very popular today. Agile methods are an outgrowth of the Agile Manifesto (`www.agilemanifesto.org`), which declares (among other things) that there's more value in working software than in the documentation created by the waterfall method and Unified Process.

  Within the category of agile methods are a variety of methods, such as XP, Scrum, and Lean. Agile methods are also iterative, but even more than in the Unified Process, a little bit of each activity is done during each iteration.

All these methods are useful, and everything I tell you in this book about developing software architecture applies to any process you use. The only differences involve when the architecture descriptions are handed off to people working on the other parts of the process.

Identifying the Problem to Be Solved

As you define your software architecture, the most important question you need to ask is: "What problem am I solving?" A major reason why software systems don't succeed is that they don't meet the needs of the customer or client who requested the software. In other words, they didn't solve the customer's or client's problem.

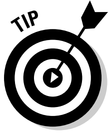
In this section, I show you how to identify the problem so that you can develop a solution that both solves the problem and meets your customer's or client's needs.

Breaking the problem into the four attributes

The problems that you solve with software architectures have four main attributes:

- **Function:** Describes the problem to be solved
- **Form:** Describes the shape of the solution and how it fits into the environment of other systems and technologies
- **Economy:** Describes how much it costs to build, operate, and maintain the solution
- **Time:** Describes how the problem is expected to change in the future

Understanding these four attributes is critical to identifying the problem to be solved.



Ask the customer what he wants in a system and why he wants it. As he explains, take notes, and map them to these problem attributes.

Ultimately, the system described by your architecture must do what the customer wants, at a cost the customer is willing to pay, and on a schedule that satisfies the customer's needs.

Developing a problem statement

A problem statement is needed to understand what to build.

To show you how to develop a problem statement, I start by walking you through the process of creating an example payroll system. Follow these steps:

1. **Establish the goals of the problem-definition process.**

   Decide how long you can spend developing the problem statement and how much detail the problem statement needs to have.

   For a payroll system, you want to identify the constraints on the solution (issues that will affect its form, economy, and time) and be sure that you understand the high-level function: to get employees paid.

2. **Gather facts.**

   In the fact-gathering steps, you work with the customer or client to understand her needs, how she's satisfying that need now, and what computing platform she expects to be used in the solution. You also identify the people and other systems, known as *actors,* that will interact with the system. Your objective is to find out as much as you can about the problem, the need, and the expectations on the system.

   For the example payroll system, you would gather facts about the number of employees, how frequently they get paid, how their pay is calculated, and what potential deductions are taken from their pay.

3. **Uncover the concepts that are essential to the solution and that will shape your architecture.**

   In this step, you look for the underlying concepts in play. You uncover assumptions, equations, regulations, process models, usage constraints, and other fundamental concepts.

   For the payroll system, you discover the equations used to compute an employee's pay and determine how irregularities from normal payment are communicated with the system.

4. **Determine what the customer or client must have to be satisfied with the solution.**

   This step involves understanding the needs and expectations of the customer or client based on the underlying concepts that you found in Step 3.

What is the minimum that the customer must have to be happy with the solution you design?

The example payroll system needs to take in each employee's hours worked, to know the base rate of pay and related deductions, and to compute payment amounts. The system also needs to print checks or in some other way make payments to the employees.

5. **Write the problem statement.**

   Based on your understanding of the problem from completing the preceding four steps, you can write a problem statement that brings in the four attributes of function, form, economy, and time (see the preceding section) in a way that explains it to the customer or client.

   For the example payroll system, the problem statement is "Compute and pay employees for work done [Function] using an interactive system for entering hours worked and for making payment through direct deposit [Form]. The solution should be available in three months [Time] for the price negotiated [Economy]."

Defining the important use cases

When you have a clear idea of what the problem is, you want to refine that definition and really zoom in on what you need to do to solve the problem. An effective way to do this is to write use cases. A *use case* describes what a person should expect to accomplish when he or she uses the system. *Actors* — the people or other systems that interact with the system being designed — are the main ingredients in use cases, and I discuss them separately later in this section.

The scenarios shown in use cases connect the different views of the architecture, showing how the parts of the architecture work together to solve the problems that you've identified by describing example usage scenarios.

Choosing the functionality to capture

You write a use case to explain how some of the system's functions work and how the system interacts with actors. Use cases can be used to explain external functionality or what goes on inside the system. The external functionality is what you want to understand at this stage of your architecture development, so concentrate on the interactions of external actors with the system. As you develop your architecture, using the method I explain in Chapter 2, the internal functions of the system become clear.

Use cases can capture large functionality, such as computing weekly payroll for all employees, or small functionality, such as validating the hours worked by a single employee. Regardless of size, however, all use cases have discrete goals — specific outcomes that they describe.

To see how use cases work, consider the simple payroll system from the previous section that computes payments due and directs those payments. Figure 1-2 shows a use-case diagram for this system and the text describing the use case. Both parts are important. This use case has one actor — the employee — who is interacting with the system to update the hours that he worked.

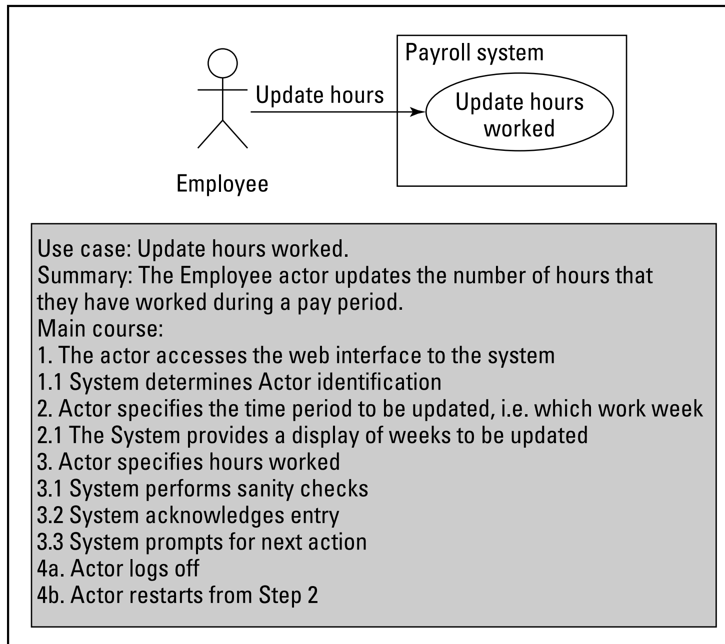


Figure 1-2: An example use-case diagram.

Use-case diagrams like the one shown in Figure 1-2 are useful for providing an overview of how the actors interact with the system and with one another.

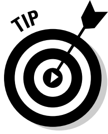
Don't try to capture all the details in a single use case. If you do, the use case will become unwieldy.

Develop the use cases a little at a time. Start by writing a high-level use case and then add more use cases that go into greater detail.

Identifying the actors

Use cases revolve around actors. Who are these actors? Here are a few definitions:

- **Actors perform the functions described in the use case.**
- **Actors play various roles: customer, user, employee, manager, payroll clerk, and so on.**



- **Actors can be involved in many use cases.** Particular actors, like the payroll clerk, can perform different functions in different use cases.
- **Actors don't need to be human; they can be other systems.**

  When an actor is a system, use a different symbol in the use-case diagram from the one you use for humans (see the next section).
- **Nonhuman actors shouldn't be internal components of the system.** Actors are people or things that interact with the system from its exterior. For the purposes of use cases, the system is a black box, and you shouldn't include its internal functioning.

Diagramming the system

Systems have multiple use cases, so a special use-case diagram provides a high-level view of how all the actors interact with the system and serves as a table of contents for the individual use cases.

Figure 1-3 shows the use-case diagram for an entire payroll system. The payroll system is in the center, surrounded by the actors. The bubbles represent the named use cases.

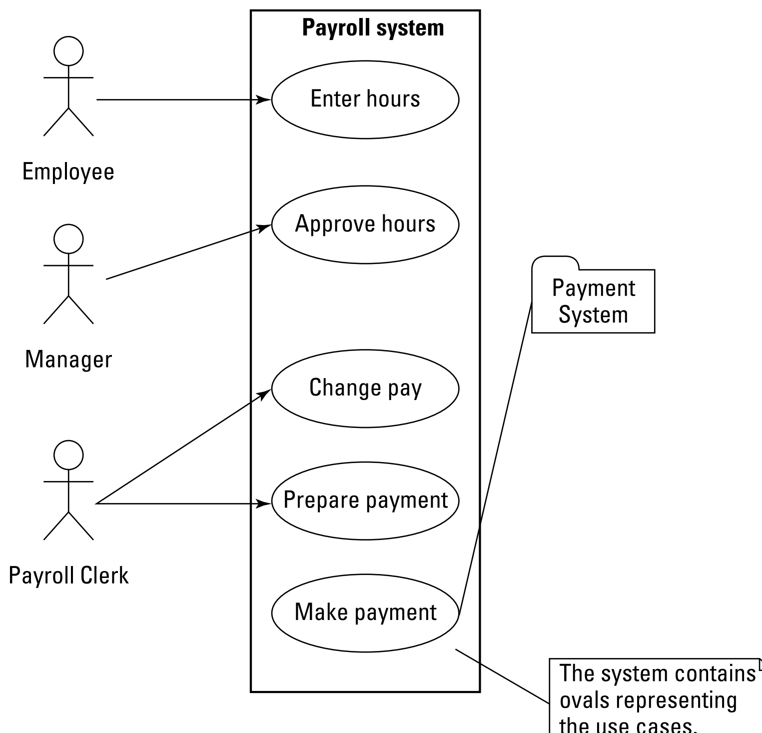


Figure 1-3: A use-case diagram for an entire architecture.

Documenting the use cases

When you begin defining your use cases, start at the overview level by identifying the most important use cases; then turn your attention to refining these use cases. Document them by using the process that follows.

These nine steps, which describe the tasks needed to develop a use case, are from *UML 2 For Dummies,* by Michael Jesse Chonoles and James A. Schardt (Wiley):

1. **Decide which use case you're going to document, and give it a name.**
2. **Sketch a diagram that shows how your actors will interact with the system.**

   For an example, refer to Figure 1-2, earlier in this chapter.
3. **Write a short summary of the use case.**

   Usually, a sentence or two is enough.
4. **Write the story of the use case.**

   The story usually begins with "The actor does *something.*"
5. **Describe the main sequence of events that will happen after the actor begins the use case.**
6. **Write down anything that must be done before the use case starts or that must be done after it ends.**
7. **Identify the other scenarios, such as error cases or alternatives.**
8. **Write the sequences of events for the alternative scenarios identified in Step 7.**
9. **Add any rules that the use case must enforce.**

   You may want to add a rule that the use case is required to validate the data input by an actor, for example.

Identifying the Requirements

When you thoroughly understand the problem to be solved, as discussed in the preceding section, you need to translate it into detailed *requirements* (the list of things that you need to include in the solution). Sometimes, you need to be formal and write down the requirements, even numbering them and tracking them through to the code. At other times, you don't need to be so formal, but you should still document the requirements. The level of detail needed in the requirements is related to the complexity of the problem and the solution; complex problems and solutions call for detailed requirements.

Architectures are created to implement and meet requirements.

You identify requirements in much the same way that you define the problem statement (refer to "Developing a problem statement," earlier in this chapter). You need to talk to the customer or client and find out what he really wants you to design and build.

Defining functional requirements

Some requirements are obvious from the customer's needs. Perhaps the customer wants the main user interface to be through a web browser, for example. Or perhaps the system needs to compute a table of values following the customer's formula, such as "compute the amount to be paid to an employee using hours worked and per-employee deductions as inputs."

Requirements like these, which define something that the system must do, are *functional requirements.* Functional requirements are represented and illustrated in use cases. When an actor interacts with the system, that interaction is made to achieve some purpose — and that purpose is the requirement.

The functional requirements show up most often in the logical view of the system (refer to "Architecture models [views]," earlier in this chapter), which shows the behavior of the individual classes.

Defining nonfunctional requirements

A system has other requirements that you won't be able to demonstrate by clicking a widget and seeing what happens. These requirements, called *nonfunctional requirements,* include things like the performance of the system, how much memory it uses, and how fast it can start.

Many lists of types of nonfunctional requirements are available, but here's the list that I like to use:

- **Changeability:** The changeability requirements all relate to how well the system can be adapted over time. The changeability-requirement family contains several subcategories:
  - Maintainability: How easy it is to maintain the system.
  - Extensibility: How easy it is to extend the system and add new functionality to it.
  - Restructuring: How easy it is to restructure the system to take advantage of new technology.
  - Portability: How easy it is to move the system to a new computing environment.

Don't allow the requirements to change too frequently — that can be a recipe for disaster. Frequent changes mean that no one will know for sure what the system is supposed to do, and they signal that the client or customer isn't sure what he or she really wants.

- **Interoperability:** The interoperability requirements describe how well the system must work with other parts of the customer's or client's computing environment.
- **Performance:** The performance requirements cover things like how fast the system must be or limits on the resources it may use.
- **Dependability:** These requirements specify how long the system must work, how secure it is, and how accurate it is. Dependability includes a large number of subcategories:
  - Reliability: How accurate the results must be and how long the system must work before it has an error.
  - Availability: The percentage of the time the system must be available for service. Availability includes fault tolerance, which specifies whether the system must tolerate faults and continue operation.
  - Maintainability: What must be designed into the system to allow it to be cared for and maintained.
  - Security: What security requirements exist for the system, what security mechanisms must be in place, what the expectations for confidentiality are, and the integrity of the system and its data.
  - Safety: Whether the public will be at risk of bodily harm from this software.

Where safety is concerned, you must look for established best practices for architecture, design, and coding within the type of system you're building. I don't talk about them in this book; you need to seek the appropriate resources.

- **Testability:** The testability requirements state what the system must do to ensure that all the requirements, both functional and nonfunctional, can be tested.
- **Reusability:** The reusability requirements specify what you need to do when designing and building the system to ensure that it can be used again. A different kind of reusability requirements specify that a certain amount of reuse be achieved within the design of the system or even that certain already-built components be used in the system.

Get SMART with your requirements

With all requirements, but especially the nonfunctional requirements, you should make the requirements SMART. This acronym reminds you that the requirements must be

- **Specific:** They describe a specific characteristic of the system.
- **Measurable:** They are testable and observable in some way.
- **Achievable:** They are realistic and can actually be achieved.
- **Relevant:** They relate to the problem that the system is supposed to solve.
- **Trackable:** They produce specific things within the architecture that you'll be able to point to later.

Here are some ways that you can identify the nonfunctional requirements:

- Find out as much as you can about the problem and how others have solved the problem.
- Extract common requirements from your reading.
- Watch how the customer uses the system that he already has, or watch him step through the process that the system will be part of.
- Listen carefully to your customer as she explains what the system will do and why it's needed.
- Ask questions!
- If you've built similar systems in the past, draw on that experience, and include the requirements that you've seen before.
- Review the problem statement with the customer so that he has the opportunity to tell you which things are important and which things are unnecessary. This review also helps refine your understanding of the problem.

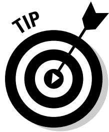
You should understand the requirements that have the biggest influence on the solution architecture first, because requirements will change. Looking at the big requirements first helps get their changes out of the way early.

The nonfunctional requirements make their appearance in both the physical and process views of the system (refer to "Architecture models [views]," earlier in this chapter). The process view addresses how the execution is distributed around the system, which may be a requirement in itself or which may be related to the performance or dependability requirements of the system. The physical view of the system also shows the nonfunctional requirements

because much of a system's dependability is tied to redundancy (how the processing is distributed to reduce the effects of single points of failure). The physical view captures the performance and scalability nonfunctional requirements through information about which processing elements can be replicated to grow the system.

Reviewing the requirements

When you're developing requirements, a variety of pitfalls can make the requirements unusable or unhelpful. The requirements may describe the problem that you want to solve or require the architecture that you want to build, rather than what the customer or client really wants and needs. Also, your requirements can omit things that the customer or client thinks is essential to the solution. Review your requirements with the customer or client to avoid these pitfalls.



Here are some things you can do to make your requirements more useful:

- **Try to identify and describe the implied or hidden requirements.** In the payroll-system example, an overtime multiplier needs to be used when the hours that any employee worked in a week exceed a given threshold.
- **Validate all your assumptions.** Perhaps you assumed that no one would receive a paycheck or a related transaction for a negative payment. This assumption may not be true, however, if wages are garnished or if many fixed deductions occur.

  Never add your own assumptions without validating them with the customer. Also make sure that you identify your assumptions — and remember that they aren't facts about the system.
- **Don't overextend assumptions.** To continue the payroll-system example, you may have assumed that everyone working more than 40 hours per week would earn base pay times some multiplier. You shouldn't keep extending this assumption by assuming, say, that the multiplier is 2. One assumption is bad enough; don't make assumptions about your assumptions.
- **Avoid indecisive specifications.** Make sure that you know whether a tax rate, for example, is *x* percent or *y* percent and that you know when it applies.
- **Avoid inconsistent or conflicting requirements.** You may have a requirement to print paychecks and another requirement to provide data for a direct deposit. Which requirement is the real requirement?

- **Fit the solution to the problem.** As you find out more about what the customer or client wants, you may see that the scope of the problem statement keeps expanding, to the point that the range of possible solutions is much larger than what the customer originally asked for (see Figure 1-4). The requirements help you see what's important and what you should really build within this range of solutions.

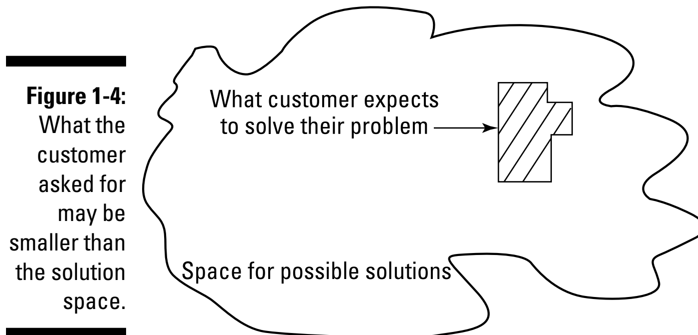


Figure 1-4: What the customer asked for may be smaller than the solution space.

Requirement do's and don'ts

Here are some good ways to ensure that your system fails:

- Don't write any requirements.
- Don't understand the usage scenarios.
- Don't understand what your customer or user really wants.
- Don't define the acceptance criteria.

By contrast, here are a few things to do (to make your requirements good and useful):

- **Describe what the system is supposed to do, and why.** Provide enough information to allow intelligent tradeoffs to be made.
- **Refrain from defining how something is to be implemented.** That definition comes into play when you are creating the architecture and design.
- **Specify technology choices only when the technology is an important aspect of the customer's problem statement.**
- **Make sure that you have all the requirements you need.** Major gaps in requirements can be critical, causing a project and/or system architecture to fail.

Choosing a Software System Style

In Chapter 2, you get down to the business of creating the actual software architecture. Before you do that, in the final step before diving in and designing the system architecture, you need to start thinking about what kind of style and shape the system should have. In this section, I highlight two aspects of system style: architectural and programming.

Architectural styles

Architectural styles define the general shape of the system. In residential housing, Cape Cod and ranch are examples of architectural styles. In software architecture, styles include Model-View-Controller and Pipes and Filters. I introduce software architecture styles in Part III.

In the different models of the architecture (such as the 4 + 1 model shown in Figure 1-1, earlier in this chapter), the views are related but also independent. You may find that you want to use a different architectural style within each view.

Programming style

You must also consider your programming style — object-based style, procedural style, or functional style, for example. Not every problem fits into every style of programming, so being familiar with multiple styles is essential to understanding the style of program you should use and choosing the right one for the problem and solution.

I won't try to explain the differences or influence your decision. Ample resources about programming in any of these styles are available, including many *For Dummies* books, and I'm sure that you have your own favorite styles. Even though this book is about patterns, however, it isn't exclusively about object-oriented programming. Patterns aren't always for objects. As you see in later chapters (specifically, Chapters 8, 23, and 24), patterns are available for a wide range of problems, not all of which relate to objects.