Applications

To build a theory, one needs to know a lot about the basic phenomena of the subject matter. We simply do not know enough about these, in the theory of computation, to teach the subject very abstractly. Instead, we ought to teach more about the particular examples we now understand thoroughly, and hope that from this we will be able to guess and prove more general principles.

MARVIN MINSKY
"Form and Content in Computer Science"

Methods are a wonderful thing, but from the perspective of the practicing engineer, the most elegant notation or process ever devised is entirely useless if it does not help us build systems for the real world. The previous chapters have been but a prelude to this section of the book, in which we now apply object-oriented analysis and design to the pragmatic construction of software systems. We have chosen a set of applications from widely varying domains, encompassing navigation, command and control, cryptanalysis, data acquisition, and Web business application design, each of which involves its own unique set of problems.

We will present the application of object-oriented analysis and design techniques by successively moving through the phases in the macro process in each of the five application chapters. The chapters progress from Inception through Elaboration to Construction. (Transition is for the most part beyond the scope of this book. However, we present some interesting post-transition considerations.) That is, each of the chapters will primarily emphasize a specific part of the macro lifecycle and the applicable analysis and design (i.e., micro process) techniques. We believe this provides a

more interesting approach than simply focusing on a single problem through all the steps of object-oriented analysis and design.

Each chapter focuses on the particular aspects of development shown here but also includes other aspects as necessary to provide context and a better understanding of the chapter's primary focus.

- Chapter 8 (satellite-based navigation) focuses on system architecture
- Chapter 9 (control system) focuses on system requirements
- Chapter 10 (cryptanalysis) focuses on analysis
- Chapter 11 (data acquisition) focuses on analysis to preliminary design
- Chapter 12 (Web modeling) focuses on detailed design and implementation

Each of these chapters could expand to fill an entire book on its own. Thus, we cannot address every phase, every activity, and every step in the process. However, we strive to address those key aspects that are most interesting and important.

The relationship of the disciplines of object-oriented analysis and design and the specific diagrams that should be used is not rigid or prescriptive. Certain diagrams are typically seen more in one phase than another. Use case diagrams are seen much more often in the early phases of a project lifecycle. Some diagrams you will rarely, if ever, encounter on a real project. However, as you will see in the following chapters, certain types of diagrams are used throughout the project lifecycle. The difference is in the level of abstraction that the diagrams capture. For example, early in the lifecycle, component diagrams may capture very large, coarse elements (e.g., systems or subsystems). Later in the lifecycle, component diagrams can be used to capture fine-grained implementation elements (e.g., software executables). You will see the refinement in the level of abstraction as you progress through the application chapters.

System Architecture: Satellite-Based Navigation

The object-oriented analysis and design principles and process presented earlier in this book, as well as the UML 2.0 notation discussed in Chapter 5, apply just as well to the development of the highest-level system architecture as to the development of software. With system architecture, though, rather than developing the structure and design of classes, we are concerned with understanding the system requirements and using that knowledge to partition the larger system into its constituent segments. However, we must remember that the concerns at this level typically are quite abstract, huge in scope and impact, and uninvolved with implementation or technology details. If we understand this and take the right steps when designing the architecture, we're more likely to create a system with long-term viability—it will be more operable, maintainable, and extensible, as it should be.

In this chapter, we show how we would approach the development of the system architecture for the hypothetical Satellite Navigation System (SNS) by logically partitioning the required functionality. To keep this problem manageable, we develop a simplified perspective of the first and second levels of the architecture, where we define the constituent segments and subsystems, respectively. In doing so, we show a representative subset of the process steps and artifacts developed, but not all of them. Showing a more complete perspective of the specification of any of these individual segments and their subsystems could easily require a complete book. However, the approach that we show could be applied more completely

across an architectural level (e.g., segment or subsystem) and through the multiple levels of the Satellite Navigation System's architecture.

We chose this domain because it is technically complex and very interesting, more so than a simple system invented solely as an example problem. Today there are two principal satellite-based navigation systems in existence, the U.S. Global Positioning System (GPS) and the Russian Global Navigation Satellite System (GLONASS). In addition, a third system called Galileo is being developed by the European Union.

8.1 Inception

The first steps in the development of the system architecture are really systems engineering steps, rather than software engineering, even for purely or mostly software systems. Systems engineering is defined by the International Council on Systems Engineering (INCOSE) as "an interdisciplinary approach and means to enable the realization of successful systems" [1]. INCOSE further defines system architecture, which is our focus here, as "the arrangement of elements and subsystems and the allocation of functions to them to meet system requirements" [2].

Our focus here is to determine *what* we must build for our customer by defining the boundary of the problem, determining the mission use cases, and then determining a subset of the system use cases by analyzing one of the mission use cases. In this process, we develop use cases from the functional requirements and document the nonfunctional requirements and constraints. But before we jump into our requirements analysis, read the sidebar to get an introduction to the Global Positioning System.

Requirements for the Satellite Navigation System

The process of building systems to help solve our customer's problems begins with determining *what* we must build. The first step is to use whatever documentation of the problem or need our customer has given us. For our system, we have been given a vision statement and associated high-level requirements and constraints.

An Introduction to the Global Positioning System

The Global Positioning System provides anyone possessing a GPS receiver with the ability to know his or her position on the earth regardless of the location, the time of day, or the weather.[1] GPS satellites, in orbits at 11,000 nautical miles above the earth, are controlled and monitored from ground stations around the world. From the launch of the first GPS satellite in 1978 to the 24th in 1994, which completed the system, GPS has been a boon to worldwide navigation [3].

Navigation has progressed from the ways the earliest people remembered and recognized landmarks as they lived their daily lives to the many technological developments on the way to GPS today. Along this path, people have used maps of the earth and stars, compasses, sextants, chronometers, and current ground-based radio navigation systems such as LORAN (long-range navigation) [4].

The GPS architecture consists of three segments: Control, User, and Space. The Control Segment is comprised of six ground stations, with the master control station located at Schriever Air Force Base in Colorado. The receivers that assist many of us in our navigation efforts constitute the User Segment, which receives position information from the 24 satellites that comprise the constellation of the Space Segment [5].

GPS receivers calculate their distance from the satellites by using time and position data broadcast by the satellites. Specifically, "If we know our exact distance from a satellite in space, we know we are somewhere on the surface of an imaginary sphere with a radius equal to the distance to the satellite radius. If we know our exact distance from two satellites, we know that we are located somewhere on the line where the two spheres intersect. And, if we take a third and a fourth measurement from two more satellites, we can find our location. The GPS receiver processes the satellite range measurements and produces its position" [6].

The Global Positioning System has numerous uses, both military and civilian. Most people are familiar with its use by military personnel for navigation on land, at sea, and in the air. It is also used on weapon systems such as the cruise missile for precise real-time navigation in support of targeting. But it's the civilian applications that have crept into many people's lives. GPS is used by emergency services to quickly provide support to people in need. It was used during the construction of the English Channel Tunnel to ensure that separate teams digging from England and France met in the middle at the precise location. It's even used in numerous personal activities such as driving, geocaching,[2] and hiking [7].

1. The Aerospace Corporation developed the *GPS Primer—A Student Guide to the Global Positioning System,* which is the source of this introductory information. Additional information can be found in the Aerospace Corporation's Summer 2002 issue of *Crosslink*, which focuses on satellite navigation and the GPS.

2. You can find the Official Global GPS Cache Hunt Site at www.geocaching.com/.

Vision:

- Provide effective and affordable Satellite Navigation System services for our customers.

Functional requirements:

- Provide SNS services
- Operate the SNS
- Maintain the SNS

Nonfunctional requirements:

- Level of reliability to ensure adequate service guarantees
- Sufficient accuracy to support current and future user needs
- Functional redundancy in critical system capabilities
- Extensive automation to minimize operational costs
- Easily maintained to minimize maintenance costs
- Extensible to support enhancement of system functionality
- Long service life, especially for space-based elements

Constraints:

- Compatibility with international standards
- Maximal use of commercial-off-the-shelf (COTS) hardware and software

Obviously, this is a highly simplified statement of requirements, but it does provide the very basic specification for a satellite-based navigation system. In practice, detailed requirements for a system as large as this come about only after the viability of a solution is demonstrated, and then only after many hundreds of person-months of analysis involving the participation of numerous domain experts and the eventual users and clients of the system. Ultimately, the requirements for a large system may encompass thousands of pages of documentation (and, hopefully, visual models), specifying not only the general behavior of the system but also intricate details such as the screen layouts to be used for human/machine interaction.

Defining the Boundaries of the Problem

Though minimal, the requirements and constraints do permit us to take an important first step in the design of the system architecture for the Satellite Navigation System—the definition of its context, as shown in Figure 8–1. This context diagram provides us with a clear understanding of the environment within which the SNS must function. Actors, representing the external entities that interact with the system, include people, other systems that provide services, and the actual envi-

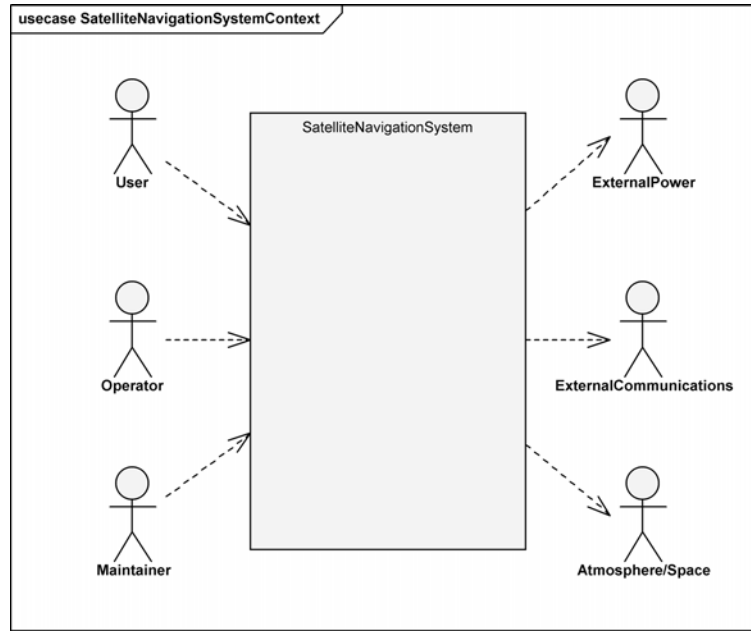

Figure 8–1 The Satellite Navigation System Context Diagram

ronment. Dependency arrows show whether the external entity is dependent on the SNS or the SNS is dependent on it.

It is quite clear that the `User`, `Operator`, and `Maintainer` actors are dependent on the SNS for its services as they use its navigation information, operate it, and maintain it, respectively. Though the Satellite Navigation System will have the capability to generate its own power as a backup for ground-based systems, primary power services will be provided by an external system, the `ExternalPower` actor. In a similar manner, we have an `ExternalCommunications` actor that provides purchased communications services to the SNS, as primary in some cases and backup to the internally provided system communications in other cases. We've prefixed the names for these two actors with "External" to clearly separate them from internal system power and communications services.

The remaining actor, `Atmosphere/Space`, may seem rather odd until we consider that it is the transmission medium for communications between the Satellite Navigation System's ground-based and space-based assets; therefore, it is a service provider. Its state certainly affects the quality of these communications. Another way to regard this actor is from the perspective of the constraint "Compatibility with international standards." Numerous national and international regulations and treaties govern satellite transmissions; thus, we have important reasons to specify this actor.

A critical point about our context diagram is the actual boundary of the system, that is, what is inside our system and what is not. Some may question our placing of the `Operator` and `Maintainer` actors outside the boundary of the `SatelliteNavigationSystem` package. By doing so, we've taken the viewpoint of a particular stakeholder, our customer, whose focus is that the system be used to provide navigation information to the user. The customer's focus is not on the broader corporate enterprise within which the SNS operates, unlike the `User` actor, who would likely regard the `Operator` and `Maintainer` as inside the system. Clearly, one's perspective is the key point here. For example, if we were providing a complete turnkey system that included operation and maintenance services, we would place the `Operator` and `Maintainer` actors inside the boundary of the `SatelliteNavigationSystem` package.

We've seen numerous variations in the presentation of a context diagram, some very elaborate and some very simple. The more elaborate ones tend to provide detailed information about the information that flows, in both directions, between the actors and the system being developed. Where a system is being developed within a more mature environment, perhaps as a replacement for an existing system, this type of information is known earlier in the development cycle, and thus some development teams choose to represent it here.

The particulars are much less important than having the development team choose a style, document it, and then follow it so clarity and understanding are ensured. However, we prefer our approach to presenting a context diagram because it simply and clearly conveys the high-level concept of the system being a container of functionality that interacts with entities in its external environment. In these interactions, the system provides services to some entities and receives services from others. This is the critical understanding that is so important in the beginning of development.

In addition to the functional requirements, we've been given high-level nonfunctional requirements that apply to portions of the functional capability or to the system as a whole. These nonfunctional requirements concern reliability, accuracy, redundancy, automation, maintainability, extensibility, and service life. Also, we see that there are some design constraints on the development of the SNS. We maintain the nonfunctional requirements and design constraints in a textual document called a *supplementary specification*; it is also used to maintain the functional requirements that apply to more than one use case. Another critical document that we must begin at this point is the *glossary*; it is important that the development team agrees on the definition of terms and then use them accordingly.

Even from these highly elided system requirements, we can make two observations about the process of developing the Satellite Navigation System.

1. The architecture must be allowed to evolve over time.
2. The implementation must rely on existing standards to the greatest extent practical.

Obviously, we cannot carry out a complete analysis or design of the Satellite Navigation System (or even the architecture) in a single chapter, much less a single book. Since our intent here is to explore how our notation and process scale up to the development of a system's architecture, we focus on the problem of designing the first and second levels of the architecture, where we define the constituent segments and subsystems, respectively. We develop these architectural levels by logically partitioning the required functionality used by the `Operator` actor. As stated in the chapter introduction, we show only a representative subset of the process steps and artifacts developed.

After reviewing both the vision and the requirements, we (the architecture team) realize that the functional requirements provided to us are really containers (packages, in the UML) for numerous mission-level use cases that define the functionality that must be provided by the Satellite Navigation System. These mission use case packages provide us a high-level functional context for the SNS, as shown in Figure 8–2. These packages contain the mission use cases that show how the users, operators, and maintainers of the SNS interact with the system to fulfill their missions. Since we are using object-oriented analysis and design techniques and the UML 2.0 notation to perform a systems engineering rather than a software engineering task, how we've used the notation in Figure 8–2 may be slightly unfamiliar. However, we believe it clearly presents the desired information and thus ensures understanding.

Determining Mission Use Cases

The vision statement for the system is rather open ended: a system to "Provide effective and affordable Satellite Navigation System services for our customers." The task of the architect, therefore, requires judicious pruning of the problem space, so as to leave a problem that is solvable. A problem such as this one could easily suffer from analysis paralysis, so we must focus on providing navigation services that are of the most general use, rather than trying to make this a navigation system that is everything for everybody (which would likely turn out to provide nothing useful for anyone). We begin by developing the mission use cases for the SNS.

Large projects such as this one are usually organized around some small, centrally located team responsible for establishing the overall system architecture, with the actual development work subcontracted out to other companies or different teams within the same company. Even during analysis, system architects usually have in

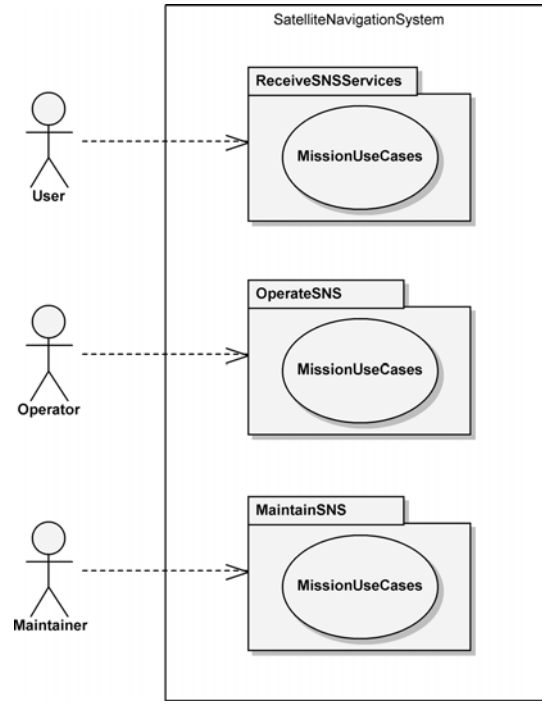

Figure 8–2 Packages for the SNS Mission Use Cases

mind some conceptual model that divides the elements of the implementation. Based on our experience in building satellite-based systems and in their operation and maintenance, we believe the highest-level logical architecture consists of four segments: Ground, Launch, Satellite, and User.

One may argue that this is design, not analysis, but we counter by saying that one must start constraining the design space at some point. Indeed, it is difficult to ascertain whether this logical architecture represents system requirements or a system design. Regardless of this issue, system architecture at this stage of development is principally object-oriented. For example, the architecture shows complex objects such as the Ground Segment and the Satellite Segment, each of which performs a major function in the system. This is just as we discussed in Chapter 4: In large systems, the objects at the highest levels of abstraction tend to be clustered along the lines of major system functions. How we identify and refine these objects during analysis is little different than how we do so during design.

Even before we have a conceptual architecture at the level of a package diagram like the one shown in Figure 8–3, we can begin our analysis by working with domain experts to articulate the primary mission use cases that detail the system's

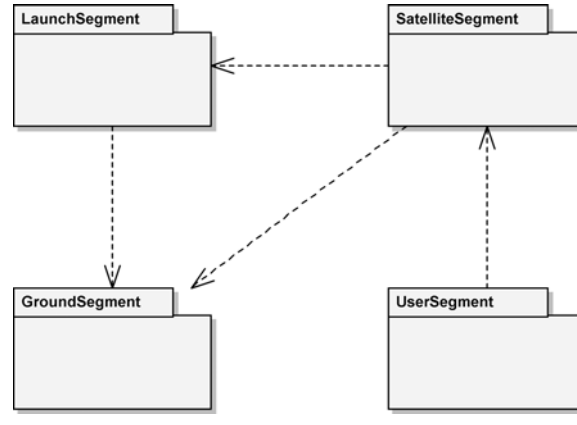

Figure 8–3 The SNS Logical Architecture

desired behavior. We say "even before" because, even though we have a notion of the architecture of the SNS, we should begin our analysis from a black-box perspective so as not to unnecessarily constrain its architecture. That is, we analyze the required functionality to determine the mission use cases for the SNS first, rather than for the individual SNS segments. Then, we allocate this use case functionality to the individual segments, in what is termed a white-box perspective of the Satellite Navigation System.

In their *Unified Modeling Language Reference Manual*, Rumbaugh, Jacobson, and Booch state that "An activity diagram is helpful in understanding the high-level execution behavior of a system, without getting involved in the internal details of message passing required by a collaboration diagram"[8].[3] Therefore, activity diagrams are our primary tool in analyzing the mission use cases and thereby illustrating the expected behavior of the system. Some development teams use sequence or communication diagrams for this purpose, but we believe those diagrams are more suitable for design efforts at lower levels within our architectural hierarchy. In our analysis, we focus only on the success scenarios of our mission use cases; the numerous alternate scenarios are left to another day.

The term *success scenario* might not be a familiar one. The well-worn ATM example helps with this explanation. One use case for the ATM is `Withdraw Cash`. This is typically what we want to do at one of these machines. In withdrawing cash, we interact with the ATM through many different steps: swipe card, enter PIN, choose withdrawal, choose amount, and so on. None of these steps embodies the end goal (withdraw cash) to us, so they are not really use cases

3. In UML 2.0, the collaboration diagram is called a communication diagram.

themselves. The `Withdraw Cash` use case (and all use cases) contains many different scenarios, each an individual path through the use case functionality. We first think about the one in which we successfully withdraw cash; hence, the term *success scenario*. This is also called the *primary scenario*. The alternate or secondary scenarios deal with the situations that typically branch off a primary scenario. For example, we're proceeding down the primary scenario of `Withdraw Cash` and get to the point of selecting the amount of cash we want. The ATM responds that we've requested more than the withdrawal amount permitted in one day. Oops! It requests that we select another amount, which we do, and then we get our cash (much less than originally desired, though). This is an illustration of a secondary scenario. It followed the path of a primary (success) scenario for several steps, veered off to deal with our amount problem, and then jumped back onto the primary scenario.

Hopefully, we've cleared rather than muddied the waters with this explanation. One more point, though—with respect to real-time systems such as the Satellite Navigation System, it is important to understand that much of its functionality is embodied within the secondary scenarios. These can be thought of as the portion of the iceberg that is below the water level yet critical to the complete and safe operation of the system. That is, the secondary scenarios are hidden in the sense that they are usually given much less attention, but they can cripple a system just as the portion of the iceberg below the water level can sink a ship. In short, our analysis must include the secondary scenarios. The amount of system functionality embodied in the secondary scenarios varies but is typically substantial in such systems. We won't consider it here—however, we must in our actual system development efforts.

Now, getting back to the task at hand, we develop the mission use cases of the `OperateSNS` mission use case package. Based on our analysis of the overall operation of the SNS, we define four corresponding mission use cases:

- `Initialize Operations`
- `Provide Normal Operations`
- `Provide Special Operations`
- `Terminate Operations`

Our specification of these mission use cases depends primarily on the domain expertise of our development team. In addition to past experience, simulations and prototypes are often invaluable tools in this analysis process. Typically, though, our analysis employs activity diagram modeling such as that which we perform to develop the system use cases in the following subsection. Figure 8–4 depicts the result of our analysis to develop the mission use cases for the

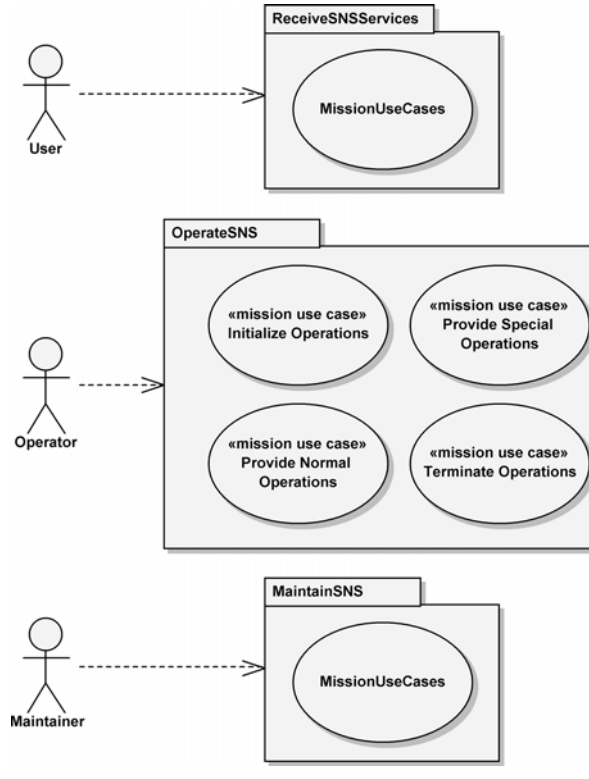

Figure 8–4 Refining the `OperateSNS` Mission Use Case Package

`OperateSNS` mission use case package. For the remainder of this chapter, our efforts focus on analyzing the `Initialize Operations` mission use case to determine the activities that the system must perform to provide the operator with the ability to initialize the operation of the Satellite Navigation System.

Determining System Use Cases

As stated previously, we develop an activity diagram of the `Initialize Operations` mission use case functionality to determine the encapsulated system use cases. In developing this activity diagram, we do not attempt to use our notion of the segments that comprise the SNS (refer back to Figure 8–3). We take this approach because we do not wish to constrain our analysis of SNS operations by presupposing possible architectural solutions to the problem at hand. We focus on the SNS as though it were a black box into which we could not peer and thus

could see only *what* services it provides, not *how* it provides those services. We are interested in the control flow across the boundary between the operator and the Satellite Navigation System as we analyze the system's high-level execution behavior.

Since we are concerned with the activities being performed by the SNS, rather than the messaging that would be represented in a communication or sequence diagram, the activity diagram is relatively simple. If we wanted to define the system activities for the entire SNS, we would perform activity diagram–based analysis for each of its mission use cases to find the myriad of activities that the Satellite Navigation System must perform to meet its requirements. Try to imagine all the activities that must be performed 24 hours a day to operate such a system. However, here we are concentrating on the `Initialize Operations` mission use case, for which we develop the activity diagram shown in Figure 8–5.

From this activity diagram, we develop the respective list of system use cases by making experienced systems engineering judgments. For example, we decide to combine the actions `Prepare for Launch` and `Launch` into one system use case, `Launch Satellite`. We determine that the remaining actions embody significant system functionality and therefore should each represent an individual system use case, giving us the system use cases for the `Initialize Operations` mission use case, as shown in Table 8–1 on page 347.

Figure 8–6 shows the system use cases of Table 8–1 in an updated use case diagram. Here we have used the `InitializeOperations` package to contain the system use cases that we developed from the `Initialize Operations` mission use case. The other three mission use cases that embody functionality for operating the SNS are shown with the keyword label of `«mission use case»`. We find this modeling approach to be useful and clear; however, each development team needs to determine and document its chosen techniques.

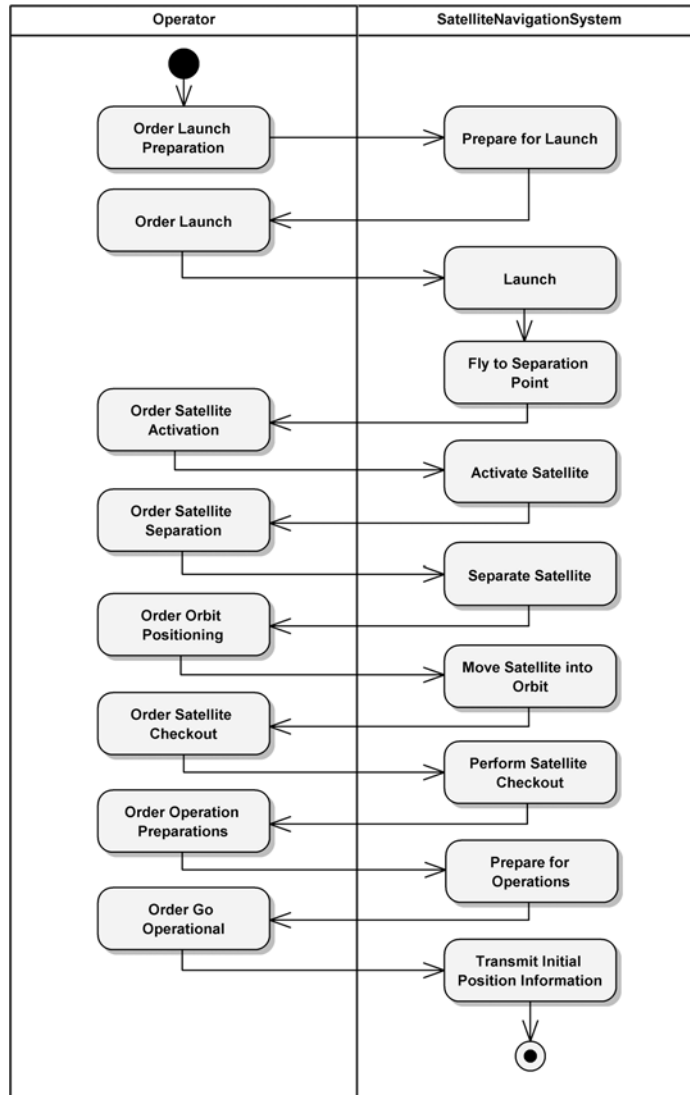

Figure 8–5 The Black-Box Activity Diagram for `Initialize Operations`

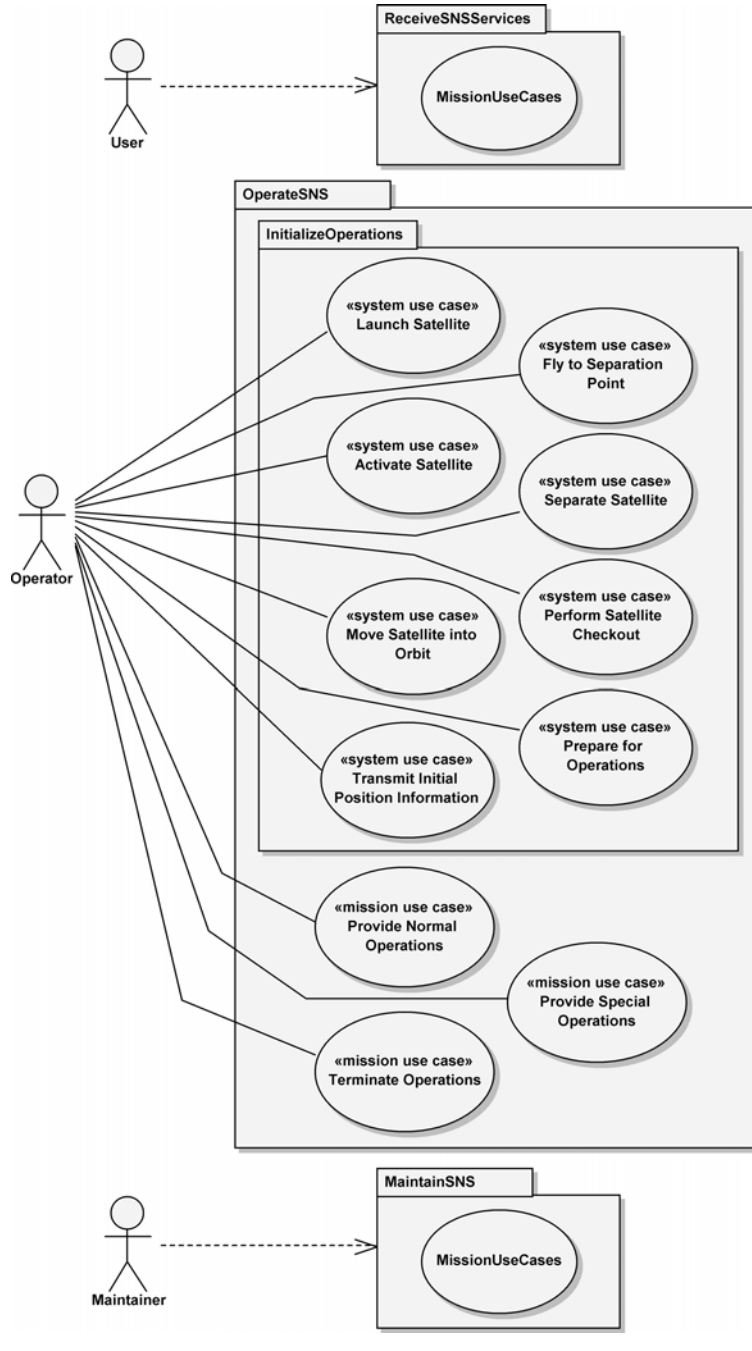

Figure 8–6 System Use Cases for `Initialize Operations`

Table 8–1 System Use Cases for *Initialize Operations*

System Use Case	Use Case Description
Launch Satellite	Prepare the launcher and its satellite payload for launch, and perform the launch.
Fly to Separation Point	Fly the launcher to the point at which the satellite payload will be separated. This involves the use and separation of multiple launcher stages.
Activate Satellite	Perform the activation of the satellite in preparation for its deployment from the launcher.
Separate Satellite	Deploy the satellite from the launcher.
Move Satellite into Orbit	Use the satellite bus propulsion capability to position the satellite into the correct orbital plane.
Perform Satellite Checkout	Perform the in-orbit checkout of the satellite's capabilities.
Prepare for Operations	Perform the final preparations prior to going operational.
Transmit Initial Position Information	Go operational and transmit initial position information to the users of the SNS.

8.2 Elaboration

Our attention turns to the system architecture that provides the foundation for realizing the requirements contained in the system use cases developed in the previous section. In the first two subsections, we introduce two architectural issues: the concerns when developing a good architecture and the activities performed when developing an architecture.

This discussion leads us into the subsequent two subsections (Validating the Proposed System Architecture, followed by Allocating Nonfunctional Requirements and Specifying Interfaces), where we perform what might be termed a macro-level analysis of the SNS system architecture. Our goal is to validate the proposed SNS architecture prior to analyzing the segments and specifying their architectures of collaborating subsystems. We use the same use case analysis techniques employed earlier to develop the system use cases, but we analyze all the use cases at the same time, rather than individually. This shortcut is perfectly valid during our behavioral prototyping but is *not* valid when actually allocating system use case functionality to the individual segments.

After the behavioral prototyping is complete, we stipulate the SNS architecture and its deployment in the next subsection. From there, we resume our actual system architectural analysis effort to decompose the Satellite Navigation System's architecture into its segments and their contained subsystems.

Developing a Good Architecture

As we discussed in Chapter 6, there are numerous methods of developing the architecture of a system. Some ways are very elegant; unfortunately, some are profoundly stupid. How do we know the difference between a good architecture and a bad one?

Good architectures tend to exhibit object-oriented characteristics. This doesn't mean, quite obviously, that as long as we use object-oriented techniques, we are assured of developing a good architecture. But, as we discussed in Chapters 1 and 2, applying the principles that underlie object-oriented decomposition tends to yield architectures that exhibit the desirable properties of organized complexity. Good architectures, whether system or software, typically have several attributes in common.

- They are constructed in well-defined layers of abstraction, each layer representing a coherent abstraction, provided through a well-defined and controlled interface, and built on equally well-defined and controlled facilities at lower levels of abstraction.
- There is a clear separation of concerns between the interface and implementation of each layer, making it possible to change the implementation of a layer without violating the assumptions made by its clients.
- The architecture is simple: Common behavior is achieved through common abstractions and common mechanisms.

Simply (or not so simply) developing a good architecture for the Satellite Navigation System is not enough; we must effectively communicate this architecture to all of its stakeholders. The Creating Architectural Descriptions sidebar explains how we may go about this task.

Defining Architectural Development Activities

The analysis and design micro process presented in Chapter 6 defines a set of development activities that are performed at each abstraction level within a system. The activities generally define the systems engineering tasks necessary to

Creating Architectural Descriptions

In the Documenting the Software Architecture sidebar presented in Chapter 6, we explained how documenting the architecture of a system has considerable value to the architects and to the other system stakeholders. We also discussed IEEE Standard 1471-2000, the IEEE Recommended Practice for Architectural Description of Software-Intensive Systems, and the 4+1 views proposed by Kruchten that present five views of software architecture: Use Case View, Logical View, Implementation View, Process View, and Deployment View.

Although IEEE Standard 1471-2000 focuses on software architecture, "it is equally applicable to any system; hence appropriate for use as a part of systems engineering to describe system architectures," according to Maier, Emery, and Hilliard [9]. In addition, they state that "A particularly important application of ANSI/IEEE 1471-2000 in systems engineering may be to reconcile and harmonize the wide array of architecture frameworks now becoming popular" [10]. They go on to compare the viewpoints for several of the more commonly used architecture frameworks: 4+1, ISO RM-ODP, DoDAF, and Zachman.

Similarly, the 4+1 views proposed by Kruchten also apply to systems engineering, according to Krikorian, who presents "Augmented 4+1 views" in which Kruchten's views are defined from the perspective of systems engineering, with appropriate activities and artifacts defined [11].

develop the system architecture for the Satellite Navigation System and are presented here, reworded for our focus.

- Identify the architectural elements at the given level of abstraction to further establish the problem boundaries and begin the object-oriented decomposition.
- Identify the semantics of the elements, that is, establish their behavior and attributes.
- Identify the relationships among the elements to solidify their boundaries and collaborators.
- Specify the interface of the elements and then their refinement in preparation for analysis at the next level of abstraction.

This set of activities makes quite clear that our primary concerns when developing the SNS architecture are the definition of its elements (segments and subsystems), their responsibilities, their collaborations, and their interfaces. These provide the architect with a framework for evolving the architecture and exploring alternative designs. One point to keep in mind is that these activities are

typically performed in parallel, rather than sequentially. For example, identifying the relationships among elements might help us to better establish their behavior and attributes. In the following subsections, we define the segments of the Satellite Navigation System, the functionality they provide, their collaborations with each other, and their interfaces.

Validating the Proposed System Architecture

We recommend that the system architects be given the opportunity to experiment with alternative system decompositions, so that we can have a fairly high level of confidence that our global design decisions are sound. This may involve modeling, simulation, or prototyping on a very large scale. These models, simulations, and prototypes can then be carried on through the maturation of this system, as vehicles for regression testing.

In this and the following subsection, we perform a macro-level analysis of the SNS system architecture to validate our assumptions and decisions before proceeding further. We want to ensure that any problems with the architecture are found now, rather than later. In the same manner that requirements changes are simpler and less expensive to accommodate earlier in the development lifecycle, so are architecture changes. We focus on the `Initialize Operations` functionality because, for example, this has been a problematic area in previous developments.

We must add a note of caution here: Though the basic techniques we use in this section and the next are very similar to those used when developing the actual architecture of the system, we apply them very differently here. We focus on drilling very quickly through several architectural levels to study some broader system concerns that we have with respect to the `Initialize Operations` mission use case. The models that we derive here are not used as part of our specification of the actual system architecture.

Far too often the initial architectural decisions are not validated because architecture teams are not aware of the utility of this step or because of the rush to move on in the development. This being the case, teams often immediately proceed to decompose the system use cases to develop segment use cases that are allocated to the segment architecture teams. The segment architecture teams then repeat the process to define subsystem use cases. Eventually, the architecture teams may attempt to recompose these use cases up through the architectural levels to determine whether everything holds together at each architectural level. Unfortunately, if it does not, it is too late. Any fixes at this late stage would likely be much more difficult, time consuming, and costly. This is why performing a macro-level analysis *before* developing the segment use cases is advantageous. This same process

should be regarded as necessary by the segment architecture teams, for the very same reasons, when they begin their analyses.

The first step is to review the results of our previous work, assess where we stand, and plan the path forward. With our domain experts, we evaluate the SNS logical architecture (refer back to Figure 8–3) and the black-box activity diagram for the `Initialize Operations` mission use case (refer back to Figure 8–5), from both functional and nonfunctional perspectives. We believe that we've captured the functionality correctly, yet we are not sure about several of the nonfunctional requirements. We have the requirement to ensure that the SNS has functional redundancy in critical system capabilities. At this level in our system architecture, we are laying out the structure of the segments, not designing their internal architectures. So, here we must ensure redundancy across segments.

To meet this requirement for functional redundancy, we choose to make two strategic system architecture decisions. First, we will have backup hardware for mission-essential equipment within the SNS Ground Segment. This equipment will be run in a hot-swappable mode where both primary and backup are active at the same time. The backup will be able to quickly replace the primary in the event of a problem that renders the primary incapable of performing its mission. Second, we will use the same hot-swappable equipment approach with the SNS Launch Segment to ensure redundant functionality. With our Satellite Segment, we will take advantage of the fact that there is redundancy across the multiple satellites in the constellation and that there will be spare satellites either on orbit or ready to be launched. This is in addition to the functional redundancy within each satellite that its designers must provide to meet the SNS nonfunctional requirement. For the User Segment, the functional redundancy requirement will be met by simply providing a replacement for the entire User Segment. As with the Satellite Segment, the designers of this segment need to focus on internal segment redundancy, as appropriate.

Beginning with the black-box activity diagram for `Initialize Operations` presented earlier in Figure 8–5, we allocate the system functionality, shown in the `SatelliteNavigationSystem` partition, to one or more of its constituent segments: Ground, Launch, Satellite, or User. Our goal is to allocate segment use cases, derived from the system use cases, to each of the segments. This way we see SNS functionality provided by a collaborative effort of its segments. If we assign use cases appropriately, the individual segments exhibit core object-oriented principles, as follows.

- *Abstraction*: Segments provide crisply defined conceptual boundaries, relative to the perspective of the viewer.
- *Encapsulation*: Segments compartmentalize their subsystems, which provide structure and behavior. Segments are black boxes to the other segments.

- *Modularity*: Segments are organized into a set of cohesive and loosely coupled subsystems.
- *Hierarchy*: Segments exhibit a ranking or ordering of abstractions.

For a system with the complexity of SNS, this part of the analysis process could easily take months to complete to any reasonable level of detail.[4] This is one of the reasons that we strongly suggest validating architectural decisions first by trying a quick-and-dirty proof-of-concept (e.g., modeling, simulation, or prototyping) to see if this part of the analysis is on the right track. The architecture team should not attempt to generate a complete list of use cases (no amount of time is sufficient) but should study some percentage of the more architecturally significant ones here.

As we walk through each of the actions shown in Figure 8–5, we must continually ask ourselves a number of questions. Which segment, or segments, should be responsible for a certain action? Does a segment have sufficient knowledge to carry out an action directed to it, or must it delegate the behavior? Is the segment trying to do too much? Is it performing actions that are not really related in some manner? What could go wrong? That is to say, what happens if certain preconditions are violated, or if postconditions cannot be satisfied?

Isn't it interesting just how similar these questions are to those we'd be asking ourselves if we were performing software engineering? Remember what we said in the introduction to this chapter?

> The object-oriented analysis and design principles and process presented earlier in this book, as well as the UML 2.0 notation discussed in Chapter 5, apply just as well to the development of the highest-level system architecture as to the development of software.

Earlier, we performed a black-box analysis of SNS system-level functionality. Now, we focus on the internal structure of the SNS—the segments that comprise this system and the functionality that each must provide collaboratively, so that the SNS is able to meet its requirements. We accomplish this task by performing a white-box analysis of the system use case functionality listed earlier in Table 8–1. We peer inside the SNS and determine *how* it provides the required services.

There is something to be aware of before we proceed; a constraint such as "Shall use the Proteus-4 launcher" would certainly impact the allocation of functional responsibility that we intend to make. To carry this thought further, when the

4. But beware of analysis paralysis: If the system analysis cycle takes longer than the window of opportunity for the business, then abandon hope, all ye who follow this path, for you will eventually be out of business.

Satellite Segment design team is designing the architecture of its segment, the team would be impacted by a constraint such as "Shall use the Gamma II(B) satellite bus."[5] We explore this concern further in the Allocation of Functionality sidebar.

Allocation of Functionality

Eventually, we must translate the system requirements into requirements for the hardware, software, and manual operation elements of the Satellite Navigation System, so that different organizations, each with different skills, can proceed in parallel to attack their particular part of the problem. During these efforts, the architecture team is always promoting and preserving the system's architectural vision.

The allocation of functionality is a concern of the system architect throughout the development because allocation can be done at any level in the abstraction, from the highest level in the system architecture to the lowest. The following list provides examples to illustrate this assertion.

- *System*: We could allocate the functionality of the entire Satellite Navigation System to another development effort. For example, a codevelopment effort could be pursued with the European Space Agency in the development of Galileo.
- *Segment*: A prime example of allocation at the segment level is subcontracting the entire launch effort. Numerous companies provide this type of service. This would mean, of course, that we would not be developing the Launch Segment but would be defining the interfaces to it with the subcontractor team.
- *Subsystem*:[6] We could envision allocation at this level in the SNS involving the utilization of a commercially available satellite bus subsystem in the development of the Satellite Segment.
- *Component*: This is the level within the SNS architecture at which we would most likely allocate requirements to hardware, software, or manual operations. For example, the User Interface Subsystem of the User Segment would likely consist of two components, one hardware (an LCD screen) and one software (used to control the User Segment).

5. A satellite bus provides infrastructure type services (e.g., power and communications) to onboard payloads that provide specialized capabilities, such as providing position information.

6. In discussing the subsystem and component levels of the SNS architecture at this time, we have given a glimpse into the future.

Previously, we listed eight SNS system use cases in Table 8–1, which we developed from the actions in the SatelliteNavigationSystem partition of Figure 8–5. If we were actually developing the system architecture, rather than performing a quick look analysis as we are here, we would develop an individual activity diagram for each of these use cases and apportion the activity diagram actions across the segments such that they exhibit the four core object-oriented principles that we discussed previously. But, to give us a broader perspective of the functionality in this macro-level analysis, instead we analyze all eight system use cases on one activity diagram. This is easily accomplished because we aren't trying to drive too much deeper into the details; rather, we are concerned with allocating portions of the use case functionality to the segments.

With the realization of our four logical SNS segments in hand (refer back to Figure 8–3), we begin our work with the domain experts to allocate the functionality denoted in the actions. If we didn't have a notion of the system architecture at this point, we could allocate the functionality to partitions with generic names such as SegmentOne, SegmentTwo, and SegmentThree. In each partition, we would then allocate the actions such that each of the segments is defined as a specialist in providing closely related capabilities, as it collaborates with the other segments to provide the Satellite Navigation System functionality—here, initializing operations. This allocation would be continued during the analysis of multiple use case scenarios to build a more complete picture of the segments' functionality, thus supporting the choice of a meaningful name for each segment.

This approach is analogous to how we would want to design good classes, as we discussed in Chapter 3. There we said that one might measure the quality of an abstraction and suggested five metrics. Two of them—coupling and cohesion—are central concerns with regard to the key abstractions of the Satellite Navigation System's architecture, that is, its segments. We are specifying the SNS segments such that they are loosely coupled; we want them to stand "alone" with only the minimal number of connections necessary to support their collaboration to provide SNS functionality.

Cohesion is the other measure by which we may judge the quality of our chosen abstractions. Cohesion measures the degree of connectivity among the elements that comprise a single segment. The least desirable form of cohesion is coincidental cohesion, in which entirely unrelated abstractions are thrown into an SNS segment. The most desirable form of cohesion is functional cohesion, in which the elements of a segment all work together to provide some well-bounded behavior.

By stepping through a few of the actions in Figure 8–5 together, we'll see how we arrived at the white-box activity diagram of Figure 8–7. The Launch Satellite system use case consisted of two actions, Prepare for Launch and Launch. It's fairly obvious that the GroundSegment and LaunchSegment should be providing this capability. The GroundSegment needs to perform its

preparations for launch and also command the LaunchSegment to do its preparations. After preparations are complete, the Operator orders the GroundSegment to launch, which then commands the LaunchSegment to do so. From the SNS system use case Launch Satellite, we have derived several actions each for the GroundSegment and the LaunchSegment. We continue this analysis process with the remaining system use cases in the Initialize Operations package to develop the complete activity diagram shown in Figure 8–7.

The white-box activity diagram for Initialize Operations presents the results of analyzing only a portion of the functionality contained within the OperateSNS mission use case package. What remains are all the preparatory activities that lead up to this point and all the activities that occur afterward, which are contained within the other three mission use cases: Provide Normal Operations, Provide Special Operations, and Terminate Operations. However, these are not our focus in this macro-level analysis. If they were, we would repeat our analysis techniques to specify this behavior and thereby develop a more complete picture of how the segments cooperate to provide the Satellite Navigation System's operational capability.

This capability would include preparatory activities such as activating the Ground and Launch Segments, checking the integrity of the satellite, and mating the satellite with the launcher. In addition to this capability, we would find that the Ground Segment performs many activities during normal operations, including the following:

- Continuously monitoring and reporting system status
- Continuously evaluating satellite flight dynamics and managing station keeping
- Monitoring for and reporting on alarms
- Managing events, including initialization and termination
- Optimizing satellite operations: estimating propellant and extending satellite life
- Recovering from power failure
- Managing satellite quality of service
- Developing operational procedures (routine and emergency)

We wouldn't be done at this point because the system functionality embodied in the MaintainSNS and ReceiveSNSServices mission use case packages of Figures 8–2 and 8–6 would also need to be analyzed to completely define the architecture of the Satellite Navigation System. However, we'll continue here with our analysis of the Initialize Operations capability for our quick look.

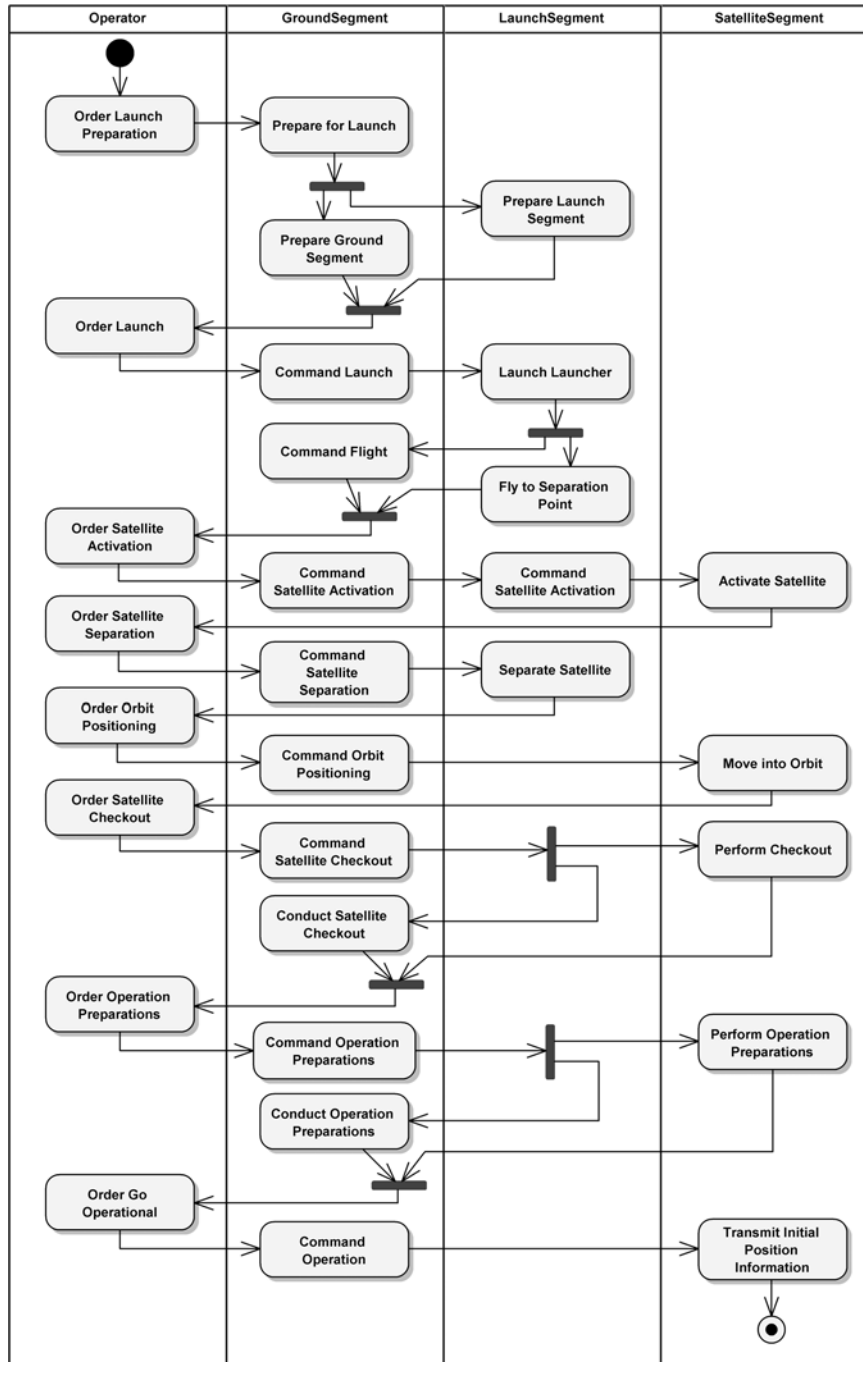

Figure 8–7 The White-Box Activity Diagram for `Initialize Operations`

The next step is to define use cases for each of the SNS segments, from the activity diagram in Figure 8–7. We do this by focusing on one partition at a time and determining which actions encompass reasonable use case functionality, by themselves or in combinations. Let's start with the `GroundSegment` partition. We decide that the first three actions—`Prepare for Launch`, `Prepare Ground Segment`, and `Command Launch`—provide the functionality for a use case we name `Control Launch`. The next action, `Command Flight`, is significant in its scope, so we define a single use case named `Control Flight` to enclose its behavior. We continue this approach for the entire `GroundSegment` partition and then repeat it for the `LaunchSegment` and `SatelliteSegment` partitions. Table 8–2 shows the resulting segment use cases and their constituent actions for the white-box `Initialize Operations` activity diagram in Figure 8–7.

Table 8–2 Segment Use Cases for *`Initialize Operations`*

SNS Segment	Segment Use Case	Segment Use Case Action
GroundSegment	Control Launch	Prepare for Launch
		Prepare Ground Segment
		Command Launch
	Control Flight	Command Flight
	Command Satellite Activation	Command Satellite Activation
	Command Satellite Separation	Command Satellite Separation
	Control Orbit Positioning	Command Orbit Positioning
	Command Satellite Checkout	Command Satellite Checkout
		Conduct Satellite Checkout
	Conduct Operation Preparations	Command Operation Preparations
		Conduct Operation Preparations
	Command Operation	Command Operation

{continued)

Table 8–2 Segment Use Cases for *Initialize Operations* *(Continued)*

SNS Segment	Segment Use Case	Segment Use Case Action
LaunchSegment	Launch	Prepare Launch Segment
		Launch Launcher
	Fly to Separation Point	Fly to Separation Point
	Command Satellite Activation	Command Satellite Activation
	Separate Satellite	Separate Satellite
SatelliteSegment	Activate Satellite	Activate Satellite
	Maneuver to Orbit	Move into Orbit
	Prepare for Operations	Perform Checkout
		Perform Operation Preparations
	Transmit Initial Position Information	Transmit Initial Position Information

Allocating Nonfunctional Requirements and Specifying Interfaces

During the analysis of the Initialize Operations functionality, we received an additional nonfunctional requirement on the Satellite Navigation System: "The time from the beginning of launch preparation to the beginning of the satellite transmitting navigation information shall be less than 7 days." Why might we be given such a requirement? Perhaps our customer doesn't want to use the approach of replacing malfunctioning satellites with on-orbit spares,[7] preferring to be able to launch a replacement satellite and have it operational within a week of need. The task we face is apportioning the 7 days (168 hours) among the use cases shown in Table 8–2 and any additional ones, such as mating the satellite

7. Replacing a malfunctioning satellite can be accomplished by using a satellite that was previously launched into space as a spare. It sits in space, like a sports player on the bench waiting to fill in for an injured player.

to the launcher, so that the entire timeline spans fewer than 168 hours, as our customer specified.

A reasonable question at this point is "How do we do this?" Well, there are two primary issues here, apportioning the nonfunctional requirement and documenting the result. Allocating the appropriate portion of the performance requirement (168 hours) to each segment use case relies a great deal on domain expertise. In addition to using the experience of the domain experts and development teams, we employ other techniques such as simulation to determine the impact of alternate allocation schemes. For our example, 48 of the 168 hours have been allocated to the `Initialize Operations` segment use cases shown in Table 8–2. The remaining 120 hours have been allocated to all the preparatory activities, which include activating the Ground Segment, activating the Launch Segment, checking the satellite integrity, and mating the satellite with the launcher.

The second issue, how to document the results, depends largely on the requirements and visual-modeling tools that the team is using and, of course, on the development process. Many tools do provide a way to document the results, but the information is usually in a requirements database that has a reference to the activities in the visual model or is buried under a tab within a properties box for an activity. While this is useful for running reports and performing statistical analysis, it doesn't provide the visual representation that we prefer, especially at this level in our development efforts. Here, we've chosen to use a table, specifically, Table 8–3, to clearly present the results of our effort to allocate the 48 hours across the segment use cases. These same techniques would be used to allocate other nonfunctional requirements across all the segment use cases that we would eventually specify.

The nonfunctional requirements allocated to a segment use case are then, at the next lower level in the architecture hierarchy, apportioned across its constituent subsystem use cases, employing the same techniques used at the segment level. Our techniques for allocating functional and nonfunctional requirements can be applied recursively from one level to the next in the architectural hierarchy—from the system to the segments, to their subsystems, and so forth.

We might then ask about potential requirements alluded to by the design constraints we've been given:

- Compatibility with international standards
- Maximal use of COTS hardware and software

The constraint "Compatibility with international standards" drove our specification of the external actor `Atmosphere/Space`, as discussed earlier. We must interact with the national and international agencies that regulate the use of the

Table 8–3 Launch Time Allocations for *Initialize Operations*[a]

SNS Segment	Segment Use Case	Allocated Time (hours:minutes)
GroundSegment	Control Launch	11:22
	Control Flight	0:17
	Command Satellite Activation	0:01
	Command Satellite Separation	0:01
	Control Orbit Positioning	0:05
	Command Satellite Checkout	16:30
	Conduct Operation Preparations	4:30
	Command Operation	0:01
LaunchSegment	Launch	11:30
	Fly to Separation Point	0:17
	Command Satellite Activation	0:01
	Separate Satellite	0:04
SatelliteSegment	Activate Satellite	0:03
	Maneuver to Orbit	13:45
	Prepare for Operations	21:29
	Transmit Initial Position Information	0:03[b]

a. If you have real-world experience in these activities, please forgive our crude allocations. Our domain experts were at lunch. Though the allocated times add up to about 80 hours, the actual clock time expended is within the 48 hours intended. This is possible because a number of actions are performed in parallel, as shown in Figure 8–7.

b. These 3 minutes denote the time it takes the Satellite System to begin transmitting position information, once commanded. Also, 40 minutes (of the 48 hours) have been allocated to the eight activities shown in the `Operator` partition in Figure 8–7.

airwaves to determine, for example, the specific frequencies at which we may communicate with the Satellite Segment, as well as the frequencies at which it may transmit position information. This means that the Ground Segment, Launch Segment (at least during the flight phase), and Satellite Segment now must fulfill the external interface responsibilities of the Satellite Navigation System. We point out these issues because in the focus on functional capability, constraints (and nonfunctional requirements) may be considered far too late in the development cycle or sometimes even overlooked.

The subject of external interfaces is one other point that we have not really touched on here, due to our focus on developing the logical architecture of the Satellite Navigation System by analyzing its functionality. The techniques to develop and document interface specifications should be familiar to those who have done any type of system or software development. The Satellite Navigation System has interfaces with those actors shown in Figure 8–1: `User`, `Operator`, `Maintainer`, `ExternalPower`, `ExternalCommunications`, and `Atmosphere/Space`. Clearly, we must perform some level of functional analysis prior to attempting the specification of the interfaces for the `User`, `Operator`, and `Maintainer` actors. In addition, human/machine interface specialists would be critical team members in this task. Interfaces to `ExternalPower` and `ExternalCommunications` actors could be specified quite early because of the standards dictating the provision of power and communications. The final external interface, the one to the `Atmosphere/Space` actor, is largely specified by national and international governments and agencies through regulations and treaties that govern satellite transmissions.

Stipulating the System Architecture and Its Deployment

The notion of the SNS logical architecture that we presented earlier in Figure 8–3 has withstood the test of our behavioral prototyping efforts. Consequently, Figure 8–3 represents the logical view of the first-level SNS architecture, as does the component diagram shown in Figure 8–8. With UML 2.0, the component diagram may be used to hierarchically decompose a system, and thus it here represents the highest-level abstraction of the Satellite Navigation System architecture, that is, its segments and their relationships. Figure 8–8 illustrates two ways to represent the interface between segments, the ball-and-socket notation (`LaunchSupport` interface) and the dashed dependency line connecting the required and provided interfaces (`PositionInformation` interfaces).

Looking back at Figure 8–1, we see that three of the system actors are not accounted for by the SNS interfaces shown in Figure 8–8: `ExternalPower`, `ExternalCommunications`, and `Atmosphere/Space`. These actors

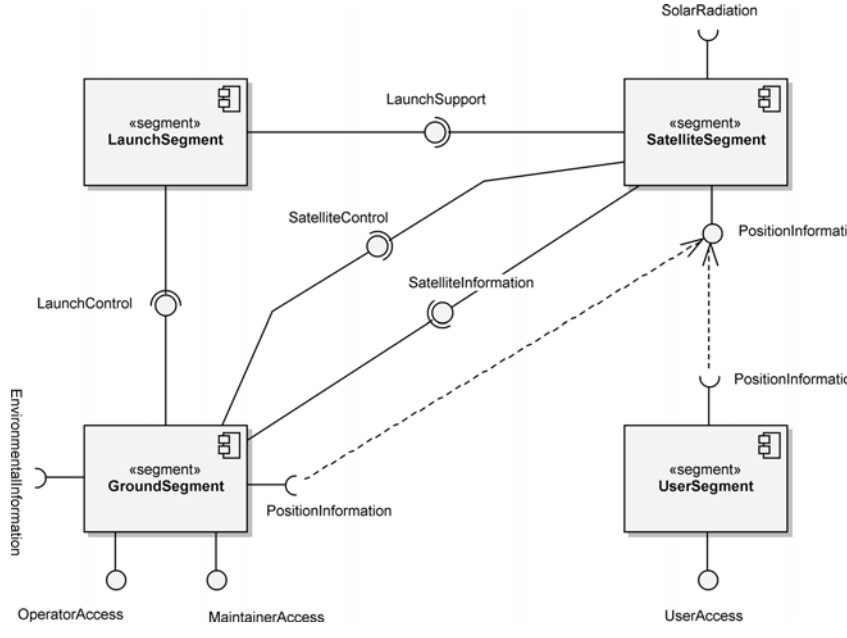

Figure 8–8 The Component Diagram for the Satellite Navigation System

provide important services to the Satellite Navigation System; however, they are not central to our focus on developing its logical architecture.

Figure 8–9 shows the deployment of the components represented in Figure 8–8 onto the architectural nodes of our system. These components are the segments of the SNS and are represented as UML 2.0 artifacts. We recognize that this is not a typical use of the notation; typically, we would deploy software artifacts (such as code, a source file, a document, or another item relating to the code) onto processing nodes. However, this diagram clearly presents the information, and some nonstandard usage is unavoidable when using the UML 2.0 notation for systems engineering. The interfaces through which the `Operator`, `Maintainer`, and `User` actors interact with the Satellite Navigation System are contained within its segments, so we've chosen to illustrate these relationships with dependencies.

Previously, we decided that our approach to providing functional redundancy was to run backups for mission-essential equipment at both the `GroundSegment` and `LaunchSegment` in a hot-swappable mode, where both primary and backup are active at the same time. In developing the SNS architecture, we've come to realize that a better approach to meeting the requirement for functional redundancy is to distribute the `GroundSegment` at two geographically dispersed sites and to do the same for the `LaunchSegment`. This protects us from

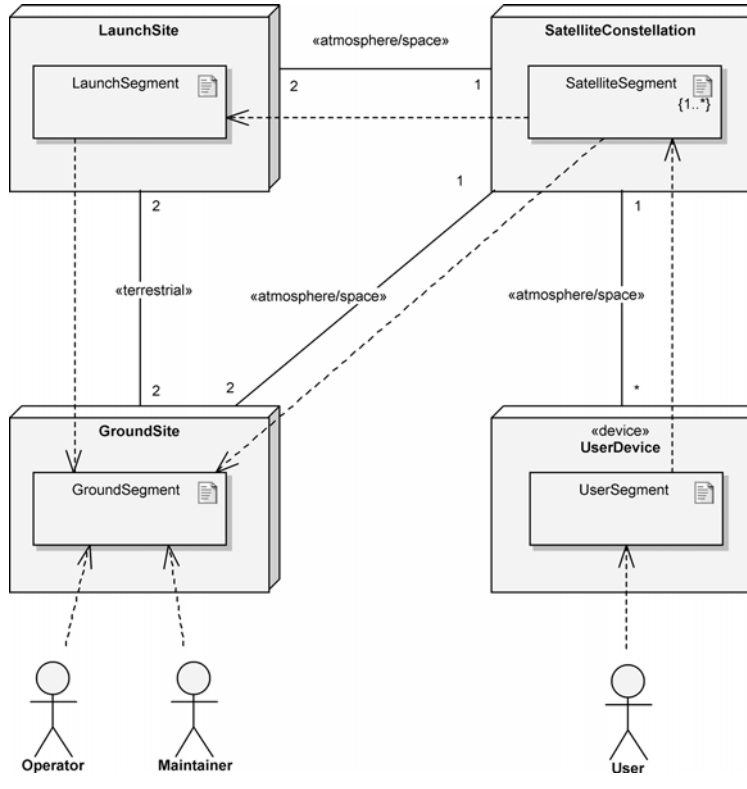

Figure 8–9 The Deployment of SNS Segments

a complete loss of segment functionality due to natural disasters, for example. This design decision is represented by the multiplicities of `2` on the communication association between the `GroundSite` and `LaunchSite` nodes. Similar to what we originally proposed with backup equipment, we now have backup sites that are prepared to assume the role of primary when commanded.

Another aspect of the SNS design that requires explanation is the `SatelliteConstellation` node and the `SatelliteSegment` artifact(s) that it hosts. The `SatelliteConstellation` node is essentially the set of Satellite Navigation System satellites and their locations in space as they provide position information to the `UserSegment` artifacts. The `SatelliteConstellation` node provides support to its hosted `SatelliteSegment` artifacts, such as gravity (an external system actor that we overlooked?) to help keep the satellites in their proper orbit and both the atmosphere and outer space to provide a communications medium for the satellites. The multiplicity of `{1..*}` on the `SatelliteSegment` artifact denotes that there is at least one satellite in the constellation. Our customer has not yet determined the coverage area for the

Satellite Navigation System.[8] When this is resolved, we will determine the actual number of satellites necessary to provide "effective and affordable Satellite Navigation System services" for the SNS users.

Decomposing the System Architecture

Now that we've validated our assumptions and decisions surrounding the Satellite Navigation System's architecture with respect to the `Initialize Operations` system use case, we can proceed with the specification of its segments and their subsystems. If we had encountered any problems, we would have modified the architecture as needed. Before we move on, we must emphasize one critical point with respect to the results of our macro-level analysis—our behavioral prototype must be discarded; it has served its intended purpose. In the same manner that prototype code is not the foundation for deliverable software, neither is our behavioral prototype the foundation for the Satellite Navigation System's architecture.

The SNS logical architecture diagram shown earlier in Figure 8–3 is useful but incomplete because each segment in this diagram is far too large to be developed by a small team of developers. We must zoom inside each of the segments and further decompose them into their nested subsystems. This is accomplished by applying the same analysis techniques—but applied more completely—that we used to prototype the Satellite Navigations System's architecture of segments for the `Initialize Operations` functionality, as depicted in the component diagram shown in Figure 8–8. These techniques are repeated through all the levels of abstraction in the Satellite Navigation System—from the system to the segments, to their subsystems, and so forth—to determine the use cases for each element at every level in the system's architecture. As we do this, the nonfunctional requirements are apportioned across the use cases, allocated to each element at every level in the system decomposition. Our analysis techniques are presented here for completeness.

1. Perform black-box analysis for each system use case to determine its actions.
2. Perform white-box analysis of these system actions to allocate them across segments.

---

8. This indecision is definitely a major source of risk (technical and nontechnical) to the program. To help our customer nail down this critical requirement, we can use simulations to determine the optimal number of satellites for a desired coverage. For those interested in this subject area, the Summer 2002 edition of the Aerospace Corporation's *Crosslink* publication (available at www.aero.org/publications/crosslink/summer2002/index.html) contains two pertinent articles: "Orbit Determination and Satellite Navigation" and "Optimizing Performance Through Constellation Management."

3. Define segment use cases from these allocated system actions.
4. Perform black-box analysis for each segment use case to determine its actions.
5. Perform white-box analysis of these segment actions to allocate them across subsystems.
6. Define subsystem use cases from these allocated segment actions.

A perspective of the black-box and white-box system analysis approach that we've used is provided in the Similar Architectural Analysis Techniques sidebar.

Normally, when applying our analysis techniques, we would complete each step, across that entire architectural level of the system, before proceeding to the next step. In other words, for step 1 we would do the black-box analysis for *all* the system use cases, *before* proceeding to step 2. Then we would do the white-box analysis of *all* the system actions *before* proceeding to step 3, and so on.

However, due to the size constraints of a chapter in a book, our example of decomposing the system architecture continues to focus on only part of the entire system—the `Launch Satellite` system use case. We make note of this so that you do not read the steps we performed (listed below) and assume that you should drill down vertically from one system use case to system activities to segment use cases to segment activities to subsystems, and so on, and then repeat for the next system use case. That would be an ineffective approach. The steps numbered above are to be applied horizontally across each architectural level of the system to provide a complete, holistic view of the system that can be validated at any point along the way.

Similar Architectural Analysis Techniques

For many years, systems engineers have been using techniques—very similar to what we describe—to analyze system functionality and allocate portions of it to the elements of a system's architecture. These techniques of black-box and white-box analysis have been used successfully to develop the complex architectural hierarchies for systems such as the Global Positioning System.

In their book *The Object Advantage*, Jacobson et al. present use cases and their application to the analysis of enterprise systems through the concepts of superordinate and subordinate use cases [12]. The IBM Rational Unified Process for Systems Engineering (RUP SE) essentially combines these approaches through systems engineering extensions to RUP. Systems engineers have been effectively employing the concepts of use cases and black-box/white-box analysis for a number of years.

We continued our analysis—not shown here—by performing the following activities:

- Performed black-box analysis for the `Launch Satellite` system use case to determine its actions
- Performed white-box analysis of these system actions to allocate them across segments
- Defined `GroundSegment` use cases from these allocated system actions
- Performed black-box analysis for the `GroundSegment`'s `Control Launch` use case to determine its actions
- Performed white-box analysis of the `GroundSegment`'s `Control Launch` actions to allocate them across its subsystems

Figure 8–10 presents the results of this analysis. We see the actions that each of the `GroundSegment` subsystems must perform as they collaborate to provide the `GroundSegment` functionality of controlling the launch. Through such analyses, we can develop the architecture of each of the Satellite Navigation System's segments. The following pages present the resulting segment architectures.

The architecture of the `GroundSegment` is composed of five subsystems: `ControlCenter`, `TT&C` (tracking, telemetry, and command), `SensorStation`, `Gateway`, and `UserInterface`, as shown in Figure 8–11. The `Control Center` subsystem essentially provides the command and control functionality for the whole of the Satellite Navigation System, with support from the `TT&C` subsystem and the `SensorStation` subsystem. The `TT&C` subsystem provides the means to monitor and control the `SatelliteSegment`, while the `SensorStation` subsystem provides position information being provided by the `SatelliteSegment` and the environmental conditions. The `Gateway` subsystem provides the means for the `ControlCenter` subsystem to communicate with the `LaunchSegment` and `SatelliteSegment` to control launch activities and satellite operations, respectively. Finally, the `UserInterface` subsystem provides `GroundSegment` functionality access to the `Operator` and `Maintainer` actors.

Figure 8–12 presents the logical architecture for the `LaunchSegment`, which is composed of three subsystems: `LaunchCenter`, `Launcher`, and `Gateway`. The `LaunchCenter` subsystem provides the command and control functionality for the `LaunchSegment`, similar to that provided by the `ControlCenter` subsystem of the `GroundSegment`. The `Launcher` subsystem provides all the capability necessary to place the `SatelliteSegment` into its initial orbit. The `Gateway` subsystem here, as in the `GroundSegment`, enables the `LaunchCenter` to receive launch control support from the `GroundSegment` and to provide launch support to the `Launcher`.

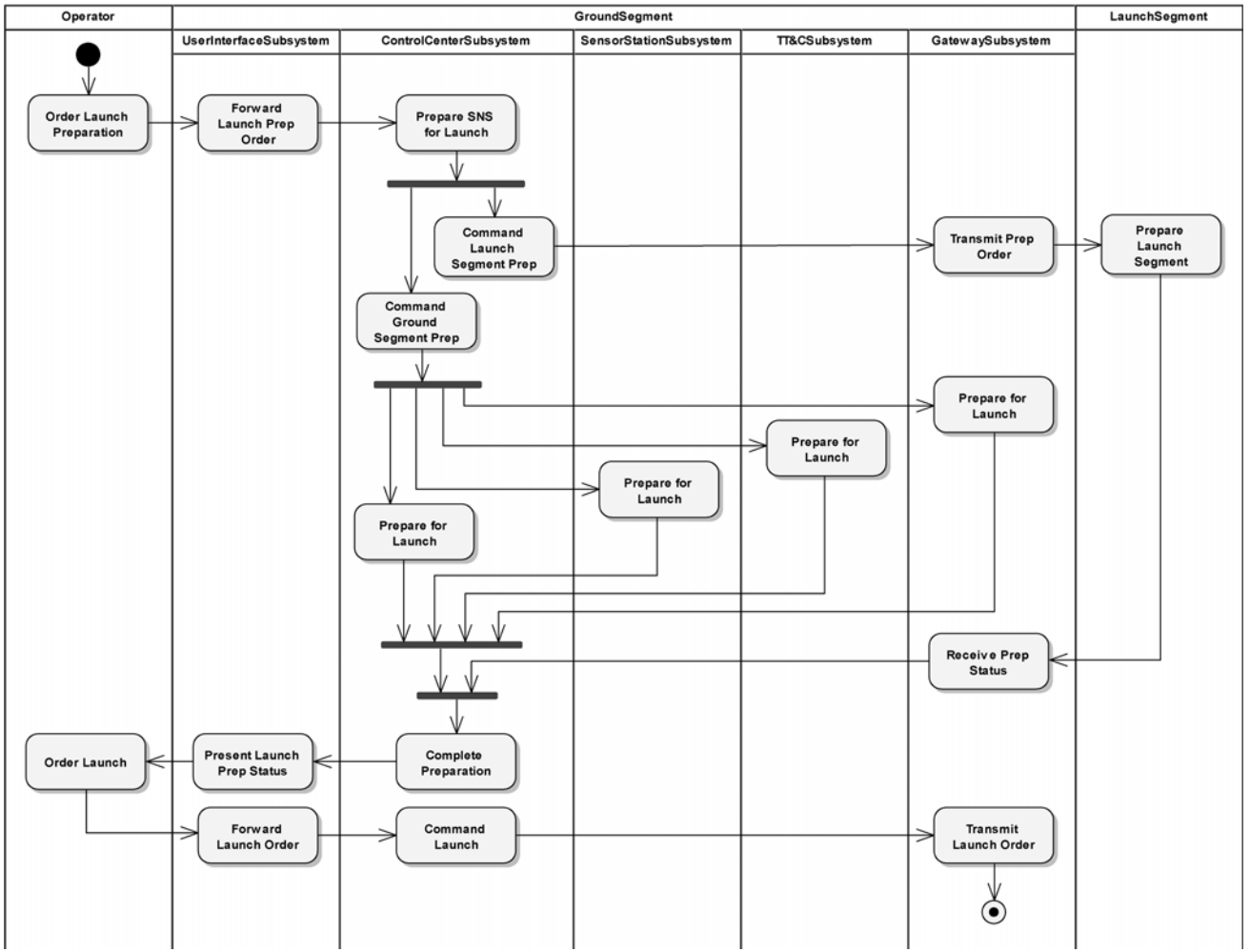

Figure 8–10 The White-Box Activity Diagram for `Control Launch`

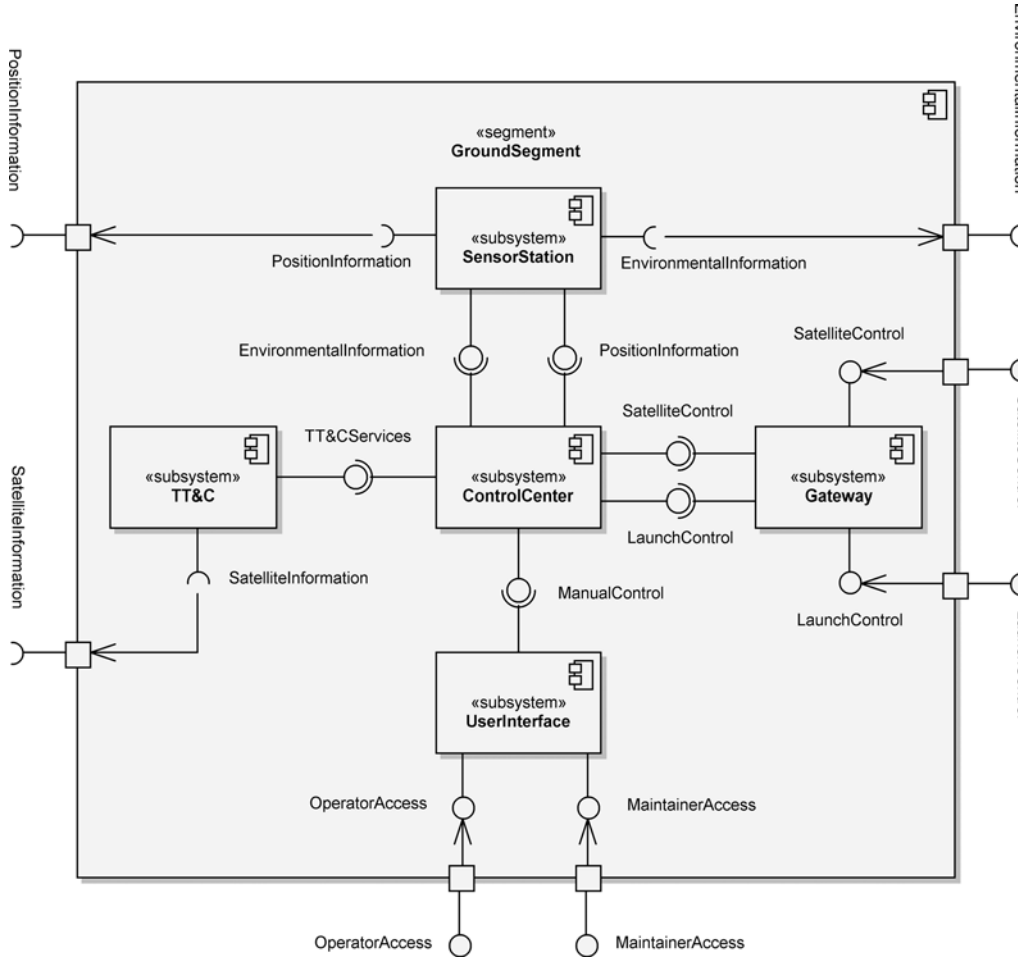

Figure 8–11 The Logical Architecture of the `GroundSegment`

The `SatelliteSegment` decomposes into two subsystems, as shown in Figure 8–13. The `SatelliteBus` subsystem provides the infrastructure support for the `NavigationPayload` subsystem. On its body structure, the `SatelliteBus` hosts equipment that provides power, attitude control, and propulsion, to name a few services. This equipment makes it possible for the `NavigationPayload` equipment (including a high-accuracy clock and position signal generation) to provide the position information to the Satellite Navigation System users.

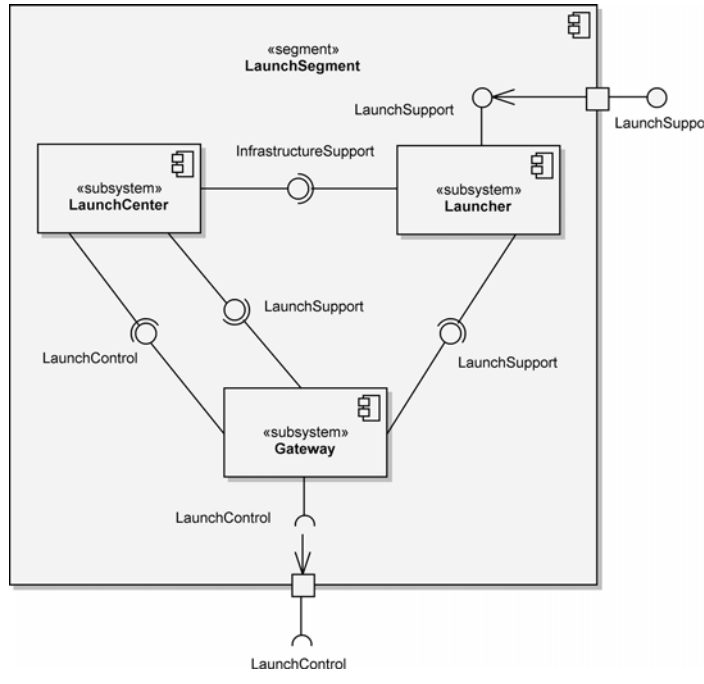

Figure 8–12 The Logical Architecture of the `LaunchSegment`

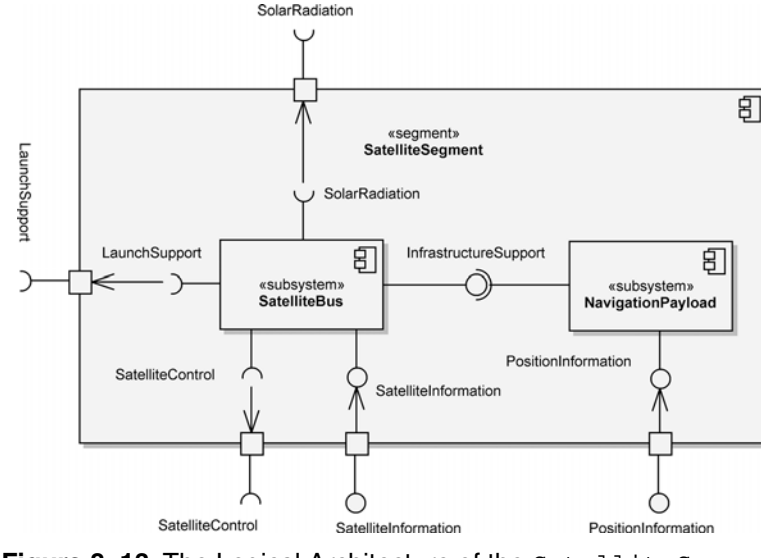

Figure 8–13 The Logical Architecture of the `SatelliteSegment`

The `UserSegment` also decomposes naturally into several subsystems, shown in Figure 8–14. The `Receiver` subsystem receives the position information from the `SatelliteSegment` and provides this as position data to the `Processor` subsystem, which translates this to navigation information for use by the `UserInterface` subsystem. The `UserInterface` provides the means for the `User` to access and make use of the `UserSegment` navigation services through a variety of specialized user interfaces using, for example, push buttons, touch screens, and audible alerts.

This design leaves us with four top-level segments, each encompassing several subsystems, to which we have allocated system functionality provided by combinations of hardware, software, and manual operations. In some cases, these allocations may be clear to the experienced system architect.

As we discussed in Chapter 7, these segments and their subsystems form the units for work assignments as well as the coarse units for configuration management and version control. Each segment or subsystem should be owned by one organization, team, or person, yet may be implemented by many more. The owner directs the detailed design and implementation of the element and manages its interface relative to other elements at the same level of abstraction. Thus, the

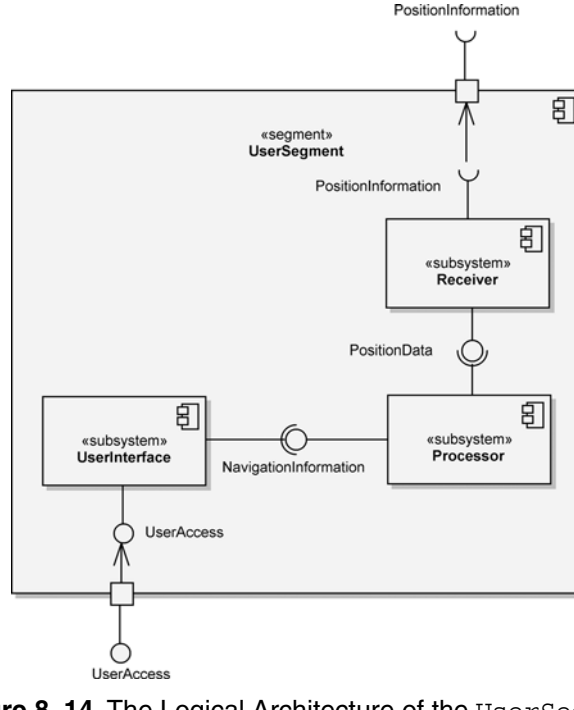

Figure 8–14 The Logical Architecture of the `UserSegment`

management of a very large development program is made possible by taking a very complex problem and decomposing it into successively smaller ones.

We have now met the goal of this chapter—we have shown that object-oriented analysis and design principles and process and the UML 2.0 notation apply just as well to the development of the highest-level system architecture as to the development of software.

8.3 Construction

At the end of the Elaboration phase, as we pointed out in Chapter 6, a stable architecture of our system should have been developed. Any modifications to the system architecture that are required as a result of activities in the Construction phase are likely limited to those of lower-level architectural elements, not the segments and subsystems of the Satellite Navigation System that have been our concern. In line with the stated intent of this chapter—to show the approach to developing the SNS system architecture by logically partitioning the required functionality to define the constituent segments and subsystems—we do not show any architectural development activities in this phase.

8.4 Post-Transition

The Satellite Navigation System's original nonfunctional requirements included two that caused us to develop a flexible architecture: extensibility and long service life. This long service life dictates, in addition to many other aspects, a design that is extensible to ensure the reliable provision of desired functionality. As there are more users of the Satellite Navigation System, and as we adapt this design to new implementations, they will discover new, unanticipated uses for existing mechanisms, creating pressure to add new functionality to the system. We now investigate how well our SNS design has met these requirements as we add new functionality and also change the system's target hardware.

Adding New Functionality

Let's consider an addition to our requirements, namely, the capability to also use the position information transmissions from other systems, such as GPS, GLONASS, and Galileo. This would add greatly to the availability and accuracy of the positioning capability of our system throughout the world. Fortunately, this

is accommodated with minimal change to the Satellite Navigation System since its impact is isolated to the User Segment. Also fortunate is the fact that our existing User Segment can be easily upgraded to provide this capability, by changing the `Receiver` subsystem and upgrading the firmware in the `Processor` subsystem. Thus, adding this functionality has done very little to our existing design. This is indeed quite common in well-structured object-oriented systems: A significant addition to the requirements for the system can be dealt with fairly easily by building new applications on existing mechanisms.

What about an even more radical change? Suppose our customer wanted to introduce the capability to support search and rescue (SAR) missions by receiving distress beacons with our Satellite Navigation System.[9] How would this new requirement affect our architecture? After analysis, we see that the greatest impact is to the Satellite Segment, but there is also some impact to the Ground Segment. This is not unexpected. If we had thought ahead to this requirement, we would have added the capability to receive these specific signals (or perhaps a range of signals), and there would be no design impact to the Satellite Segment, just the operational impact of using the functionality. Otherwise, we would need to add an additional subsystem to the Satellite Segment that would provide this capability. With respect to the Ground Segment, the impact is merely being able to relay information about the reception of the distress beacon to the appropriate civil authorities. This would likely involve minor software or operational modifications to the `ControlCenter` and `TT&C` subsystems.

Again, this large change would have minimal impact to the overall SNS architecture. Unfortunately, the difficulty of making modifications to space-based assets forces the system architects to be very far-reaching in their vision for the system. But even in the worst-case situation here, we would be able to add the SAR capability to Satellite Segment elements being developed in the future, with minimal impact on the existing architecture or functionality.

Changing the Target Hardware

Hardware technology is still moving at a faster pace than our ability to generate software. Furthermore, it is likely that a number of political and historical reasons will cause us to make certain hardware and software choices early in the develop-

9. The Galileo program is adding this type of capability from the outset to assist the Cospas-Sarsat Program (www.cospas-sarsat.org/) in its mission of supporting SAR missions throughout the world. In fact, the Galileo program believes that its contribution to this effort will provide near real-time acquisition of distress beacons and location to within several meters.

ment process that we may later regret.[10] For this reason, the target hardware for large systems becomes obsolete far earlier than does its software.

For example, after several years of operational use, we might decide we need to replace the entire `ControlCenter` subsystem of the Ground Segment. How might this affect our existing architecture? If we have kept our subsystem interfaces at a high level of abstraction during the evolution of our system, this hardware change would affect our software in only minimal ways. Since we chose to encapsulate all design decisions regarding the specifics of the `ControlCenter` subsystem, no other subsystem was ever defined to depend on the specific characteristics of a given workstation, for example; the subsystem encapsulates all such hardware secrets. This means that the behavior of workstations is hidden in the `ControlCenter` subsystem. Thus, this subsystem acts as an abstraction firewall, which shields all other clients from the intricacies of our particular computing technology.

In a similar fashion, a radical change in telecommunications standards would affect our implementation, but only in limited ways. Specifically, our design ensures that only the `Gateway` subsystem knows about network communications. Thus, even a fundamental change in networking would never affect any higher-level client; the `Gateway` subsystem shields them from the perversity of the real world.

None of the changes we have introduced rends the fabric of our existing architecture. This is indeed the ultimate mark of a well-architected, object-oriented system.

10. For example, our project might have chosen a particular hardware or software product from a third-party vendor, only to later discover that the product didn't live up to its promises. Even worse, we might find that the only supplier of a critical product went out of business. In such cases, the project manager usually has one of two choices: (1) run screaming into the night, or (2) choose another product, and hope that the system's architecture is resilient enough to accommodate the change. The use of object-oriented analysis and design helps us to achieve (2), although it is sometimes still very satisfying to carry out (1).

This page intentionally left blank

Control System: Traffic Management

The economics of software development have progressed to the point where many more kinds of applications are now automated than ever before, ranging from embedded microcomputers that control a myriad of automobile functions to tools that eliminate much of the drudgery associated with producing an animated film to systems that manage the distribution of interactive video services to millions of consumers. The distinguishing characteristic of all these larger systems is that they are extremely complex. Building systems so that their implementation is small is certainly an honorable task, but reality tells us that certain large problems demand large implementations. For some massive applications, it is not unusual to find software development organizations that employ several hundred programmers who must collaborate to produce millions of lines of code against a set of requirements that are guaranteed to be unstable during development. Such projects rarely involve the development of single programs; they more often encompass multiple, cooperative programs that must execute across a distributed target system consisting of many computers connected to one another in a variety of ways. To reduce development risk, such projects usually involve a central organization that is responsible for systems architecture and integration; the remaining work may be subcontracted to other companies or to other in-house organizations. Thus, the development team as a whole never assembles as one; it is typically distributed over space and—because of the personnel turnover common in large projects—over time.

Developers who are content with writing small, stand-alone, single-user, window-based tools may find the problems associated with building massive applications staggering—so much so that they view it as folly even to try. However, the actuality of the business and scientific world is such that

complex software systems must be built. Indeed, in some cases, it is folly not to try. Imagine using a manual system to control air traffic around a major metropolitan center or to manage the life-support system of a manned spacecraft or the accounting activities of a multinational bank. Successfully automating such systems not only addresses the very real problems at hand but also leads to a number of tangible and intangible benefits, such as lower operational costs, greater safety, and increased functionality. Of course, the operative word here is *successfully*. Building complex systems is plain hard work and requires the application of the best engineering practices we know, along with the creative insight of a few great designers.

This chapter tackles the development of such a problem.

9.1 Inception

To most people living in the United States, trains are an artifact of an era long past; in Europe and in many parts of Asia, the situation is entirely the opposite. Trains are an essential part of their transportation networks; tens of thousands of kilometers of track carry people and goods daily, both within cities and across national borders. In all fairness, trains do provide an important and economical means of transporting goods within the United States. Additionally, as major metropolitan centers grow more crowded, light rail transport is increasingly providing an attractive option for easing congestion and addressing the problems of pollution from internal combustion engines.

Still, railroads are a business and consequently must be profitable. Railroad companies must delicately balance the demands of frugality and safety and the pressures to increase traffic against efficient and predictable train scheduling. These conflicting needs suggest an automated solution to train traffic management, including computerized train routing and monitoring of all elements of the train system. Such automated and semiautomated train systems exist today in Sweden, Great Britain, West Germany, France, Japan [1], Canada, and the United States. The motivation for each of these systems is largely economic and social: Lower operating costs and more efficient use of resources are the goals, with improved safety as an integral by-product.

In this section, we begin our analysis of the fictitious Train Traffic Management System (TTMS) by specifying its requirements and the system use cases that further describe the required functionality.

Requirements for the Train Traffic Management System

Our experience with developing large systems has been that an initial statement of requirements is never complete, often vague, and always self-contradictory. For these reasons, we must consciously concern ourselves with the management of uncertainty during development, and therefore we strongly suggest that the development of such a system be deliberately allowed to evolve over time in an incremental and iterative fashion. As we pointed out in Chapter 6, the very process of development gives both users and developers better insight into what requirements are really important—far better than any paper exercise in writing requirements documents in the absence of an existing implementation or prototype. Also, since developing the software for a large system may take several years, software requirements must be allowed to change to take advantage of rapidly changing hardware technology.[1] It is undeniably futile to craft an elegant software architecture targeted to a hardware technology that is guaranteed to be obsolete by the time the system is fielded. This is why we suggest that, whatever mechanisms we craft as part of our software architecture, we should rely on existing standards for communications, graphics, networking, and sensors. For truly novel systems, it is sometimes necessary to pioneer new hardware or software technology. This adds risk to a large project, however, which already involves a customarily high risk. Software development clearly remains the technology of highest risk in the successful deployment of any large automated application, and our goal is to limit this risk to a manageable level, not to increase it.

This is a very large and highly complex system that in reality would not be specified by simple requirements. However, for this chapter, the requirements that follow will suffice for the purposes of our analysis and design effort. In the real world, a problem such as this could easily suffer from analysis paralysis because there would be many thousands of requirements, both functional and nonfunctional, with a myriad of constraints. Quite clearly, we would need to focus our efforts on the most critical elements and prototype candidate solutions within the operational context of the system under development.

---

1. In fact, for many such systems of this complexity, it is common to have to deal with many different kinds of computers. Having a well-thought-out and stable architecture mitigates much of the risk of changing hardware in the middle of development, an event that happens all too often in the face of the rapidly changing hardware business. Hardware products come and go, and therefore it is important to manage the hardware/software boundary of a system so that new products can be introduced that reduce the system's cost or improve its performance, while at the same time preserving the integrity of the system's architecture.

The Train Traffic Management System has two primary functions: train routing and train systems monitoring. Related functions include traffic planning, failure prediction, train location tracking, traffic monitoring, collision avoidance, and maintenance logging. From these functions, we define eight use cases, as shown in the following list.

- `Route Train`: Establish a train plan that defines the travel route for a particular train.
- `Plan Traffic`: Establish a traffic plan that provides guidance in the development of train plans for a time frame and geographic region.
- `Monitor Train Systems`: Monitor the onboard train systems for proper functioning.
- `Predict Failure`: Perform an analysis of train systems' condition to predict probabilities of failure relative to the train plan.
- `Track Train Location`: Monitor the location of trains using TTMS resources and the Navstar Global Positioning System (GPS).
- `Monitor Traffic`: Monitor all train traffic within a geographic region.
- `Avoid Collision`: Provide the means, both automatic and manual, to avoid train collisions.
- `Log Maintenance`: Provide the means to log maintenance performed on trains.

These use cases establish the basic functional requirements for the Train Traffic Management System, that is, they tell us *what* the system must do for its users. In addition, we have nonfunctional requirements and constraints that impact the requirements specified by our use cases, as listed here.

Nonfunctional requirements:

- Safely transport passengers and cargo
- Support train speeds up to 250 miles per hour
- Interoperate with the traffic management systems of operators at the TTMS boundary
- Ensure maximum reuse of and compatibility with existing equipment
- Provide a system availability level of 99.99%
- Provide complete functional redundancy of TTMS capabilities
- Provide accuracy of train position within 10.0 yards
- Provide accuracy of train speed within 1.5 miles per hour
- Respond to operator inputs within 1.0 seconds
- Have a designed-in capability to maintain and evolve the TTMS

Constraints:

- Meet national standards, both government and industry
- Maximize use of commercial-off-the-shelf (COTS) hardware and software

Now that we have our core requirements defined, at least at a very high level, we must turn our attention to understanding the users of the Train Traffic Management System. We find that we have three types of people who interact with the system: `Dispatcher`, `TrainEngineer`, and `Maintainer`. In addition, the Train Traffic Management System interfaces with one external system, `Navstar GPS`. These actors play the following roles within the TTMS.

- `Dispatcher` establishes train routes and tracks the progress of individual trains.
- `TrainEngineer` monitors the condition of and operates the train.
- `Maintainer` monitors the condition of and maintains train systems.
- `Navstar GPS` provides geolocation services used to track trains.

Determining System Use Cases

Figure 9–1 shows the use case diagram for the Train Traffic Management System. In it, we see the system functionality used by each of the actors. We also see that we have «`include`» and «`extend`» relationships used to organize relationships between several of the use cases. The functionality of the use case `Monitor Train Systems` is extended by the use case `Predict Failure`. During the course of monitoring systems, a failure prediction analysis (`condition: {request Predict Failure}`) can be requested for a particular system that is operating abnormally or may have been flagged with a yellow condition indicating a problem requiring investigation. This occurs at the `Potential Failure` extension point.

The functionality of the `Monitor Traffic` use case is also extended, by that of the `Avoid Collision` use case. Here, when monitoring train traffic, an actor has optional system capability to assist in the avoidance of a collision—at the `Potential Collision` extension point. This assistance can support both manual and automatic interventions. `Monitor Traffic` always includes the functionality of the `Track Train Location` use case to have a precise picture of the location of all train traffic. This is accomplished by using both TTMS resources and the Navstar GPS.

We may specify the details of the functionality provided by each of these use cases in textual documents called use case specifications. We have chosen to focus on the two primary use cases, `Route Train` and `Monitor Train`

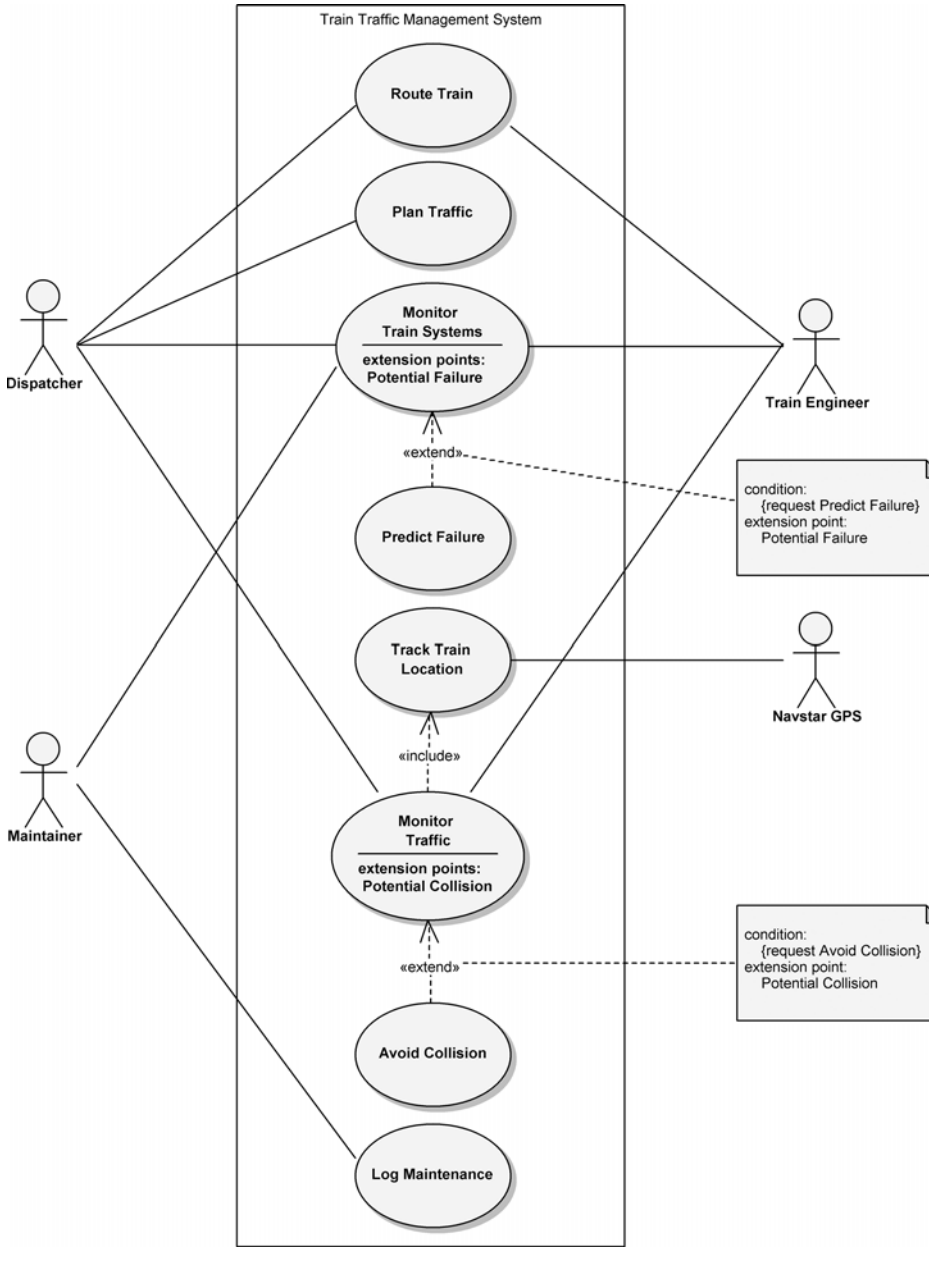

Figure 9–1 The Use Case Diagram for the Train Traffic Management System

`Systems`, in the following use case specifications. The format of the use case specification is a general one that provides setup information along with the primary scenario and one or more alternate scenarios.

It should be noted that these use case specifications focus on the boundary-level interaction between the users of the system and the Train Traffic Management System itself. This perspective is often referred to as a black-box view since the internal functioning of the system is not seen externally. This view is used when we are concerned with *what* the system does, not *how* the system does it.

Use case name: `Route Train`

Use case purpose: The purpose of this use case is to establish a train plan that acts as a repository for all the pertinent information associated with the route of one particular train and the actions that take place along the way.

Point of contact: Katarina Bach

Date modified: 9/5/06

Preconditions: A traffic plan exists for the time frame and geographic region (territory) relevant to the train plan being developed.

Postconditions: A train plan has been developed for a particular train to detail its travel route.

Limitations: Each train plan will have a unique ID within the system. Resources may not be committed for utilization by more than one train plan for a particular time frame.

Assumptions: A train plan is accessible by dispatchers for inquiry and modification and accessible by train engineers for inquiry.

Primary scenario:

A. The Train Traffic Management System (TTMS) presents the dispatcher with a list of options.

B. The dispatcher chooses to develop a new train plan.

C. The TTMS presents the template for a train plan to the dispatcher.

D. The dispatcher completes the train plan template, providing information about locomotive ID(s), train engineer(s), and waypoints with times.

E. The dispatcher submits the completed train plan to the TTMS.

F. The TTMS assigns a unique ID to the train plan and stores it. The TTMS makes the train plan accessible for inquiry and modification.

G. This use case ends.

Alternate scenarios:

Condition triggering an alternate scenario:

Condition 1: Develop a new train plan, based on an existing one.

B1. The dispatcher chooses to develop a new train plan, based on an existing one.

B2. The dispatcher provides search criteria for existing train plans.

B3. The TTMS provides the search results to the dispatcher.

B4. The dispatcher chooses an existing train plan.

B5. The dispatcher completes the train plan.

B6. The primary scenario is resumed at step E.

Condition triggering an alternate scenario:

Condition 2: Modify an existing train plan.

B1. The dispatcher chooses to modify an existing train plan.

B2. The dispatcher provides search criteria for existing train plans.

B3. The TTMS provides the search results to the dispatcher.

B4. The dispatcher chooses an existing train plan.

B5. The dispatcher modifies the train plan.

B6. The dispatcher submits the modified train plan to the TTMS.

B7. The TTMS stores the modified train plan and makes it accessible for inquiry and modification.

B8. This use case ends.

Use case name: `Monitor Train Systems`

Use case purpose: The purpose of this use case is to monitor the onboard train systems for proper functioning.

Point of contact: Katarina Bach

Date modified: 9/5/06

Preconditions: The locomotive is operating.

Postconditions: Information concerning the functioning of onboard train systems has been provided.

Limitations: None identified.

Assumptions: Monitoring of onboard train systems is provided when the locomotive is operating. Audible and visible indications of system problems, in addition to those via video display, are provided.

Primary scenario:

A. The Train Traffic Management System (TTMS) presents the train engineer with a list of options.

B. The train engineer chooses to monitor the onboard train systems.

C. The TTMS presents the train engineer with the overview status information for the train systems.

D. The train engineer reviews the overview system status information.

E. This use case ends.

Alternate scenarios:

Condition triggering an alternate scenario:

Condition 1: Request detailed monitoring of a system.

E1. The train engineer chooses to perform detailed monitoring of a system that has a yellow condition.

E2. The TTMS presents the train engineer with the detailed system status information for the selected system.

E3. The train engineer reviews the detailed system status information.

E4. The primary scenario is resumed at step B..

Extension point—`Potential Failure`**:**

Condition 2: Request a failure prediction analysis for a system.

E3-1. The train engineer requests a failure prediction analysis for a system.

E3-2. The TTMS performs a failure prediction analysis for the selected system.

E3-3. The TTMS presents the train engineer with the failure prediction analysis for the system.

E3-4. The train engineer reviews the failure prediction analysis.

E3-5. The train engineer requests that the TTMS alert the maintainer of the system that might fail.

E3-6. The TTMS alerts the maintainer of that system.

E3-7. The maintainer requests the failure prediction analysis for review.

E3-8. The TTMS presents the maintainer with the failure prediction analysis.

E3-9. The maintainer reviews the analysis and determines that the yellow condition is not severe enough to warrant immediate action.

E3-10. The maintainer requests that the TTMS inform the train engineer of this determination.

E3-11. The TTMS provides the train engineer with the determination of the maintainer.

E3-12. The train engineer chooses to perform detailed monitoring of the selected system.

E3-13. The alternate scenario is resumed at step E3.

Even though the requirements for the Train Traffic Management System are very simplified, we still have not completely specified them, and they are somewhat vague. This is not unlike what we've encountered while developing large, complex systems in the real world. As we've discussed previously, effectively managing ever-changing requirements is critical to having a successful development process, which we should all define as providing the right functionality, on time, and within budget. But don't think that our goal is to stop requirements from changing; we can't and we shouldn't want to do this. We can understand this if we focus on the rapid pace of functional enhancements made to hardware technology that, usually along with decreased cost, provide ever more solutions to our software development problems. Just look at the incredibly capable and complex software that can be run on today's personal computers, with their processors running at multigigahertz speeds and sporting gigabytes of random access memory (RAM).

So, how do we accommodate changing requirements, especially over development time frames that may encompass several years? We've found that using an iterative and incremental development process is one of the key means to managing the risks associated with changing requirements in such a large automated system. Another is designing an architecture that remains flexible throughout the development. Yet another is maximizing the use of COTS hardware and software, as one of the TTMS constraints directs us to do. As we proceed through this chapter, our prime focus will be on developing an architecture that can accommodate change.

9.2 Elaboration

Our attention now turns to developing the overall architecture framework for the Train Traffic Management System. We begin by analyzing the required system functionality that leads us into the definition of the TTMS architecture. From there, we begin our transition from systems engineering to the disciplines of hardware and software engineering. We conclude this section by describing the key abstractions and mechanisms of the TTMS.

Analyzing System Functionality

Now that the requirements for the Train Traffic Management System have been specified, our focus turns to *how* the system's aggregate parts provide this required functionality. This perspective is often referred to as a white-box view since the internal functioning of the system is seen externally. We use activity diagrams to analyze the various use case scenarios to develop this further level of detail.

Let's begin by looking at Figure 9–2, which analyzes the primary scenario of the `Route Train` use case. This activity diagram is relatively straightforward and follows the course of the use case scenario. Here we see the interaction of the `Dispatcher` actor and the `RailOperationsControlSystem`, which we've designated as the primary command and control center for the TTMS, as the `Dispatcher` creates a new `TrainPlan` object.

When determining the constituent elements of the TTMS, we must of course consider the requirements, both functional and nonfunctional, and the constraints. But we also have two competing technical concerns: the desire to encapsulate abstractions and the need to make certain abstractions visible to other elements. In other words, we must strive to design elements that are cohesive (by grouping logically related abstractions) and loosely coupled (by minimizing the dependencies among elements). Therefore, we define modularity as the property of a system that has been decomposed into a set of cohesive and loosely coupled elements.

In contrast to Figure 9–2, the activity diagram of Figure 9–3 is a bit more complicated because we've illustrated the first alternate scenario of the `Monitor Train Systems` use case, where the `TrainEngineer` chooses to perform a detailed monitoring of the `LocomotiveAnalysisandReporting` system, which has a yellow condition. Here we see that the constituent elements of the TTMS providing this capability are the `OnboardDisplay` system, the `LocomotiveAnalysisandReporting` system, the `EnergyManagement` system, and the `DataManagement` unit.

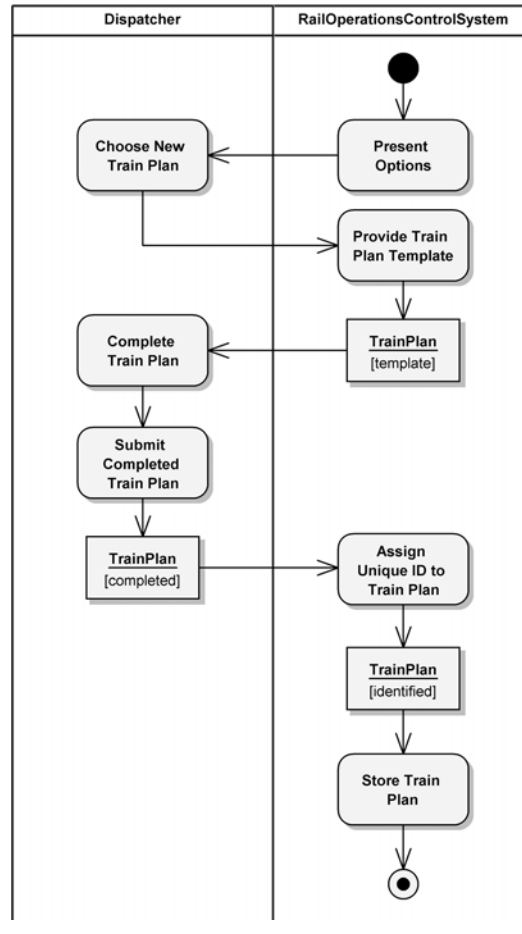

Figure 9–2 The `Route Train` Primary Scenario

We see that the `OnboardDisplay` system is the interface between the `TrainEngineer` and the TTMS. As such, it receives the `TrainEngineer`'s request to monitor the train systems and then requests the appropriate data from each of the other three systems. The overview level of status information is provided to the `TrainEngineer` for review. At this point, the `TrainEngineer` could remain at the overview level, which would end the primary scenario. In the alternate scenario, however, the `TrainEngineer` requests a more detailed review from the `LocomotiveAnalysisandReporting` system because it has presented a yellow condition indicating some type of problem that requires attention. In response, the `OnboardDisplay` system retrieves the detailed data from the system for presentation. After reviewing this information, the `TrainEngineer` returns to monitoring the overview level of system status information.

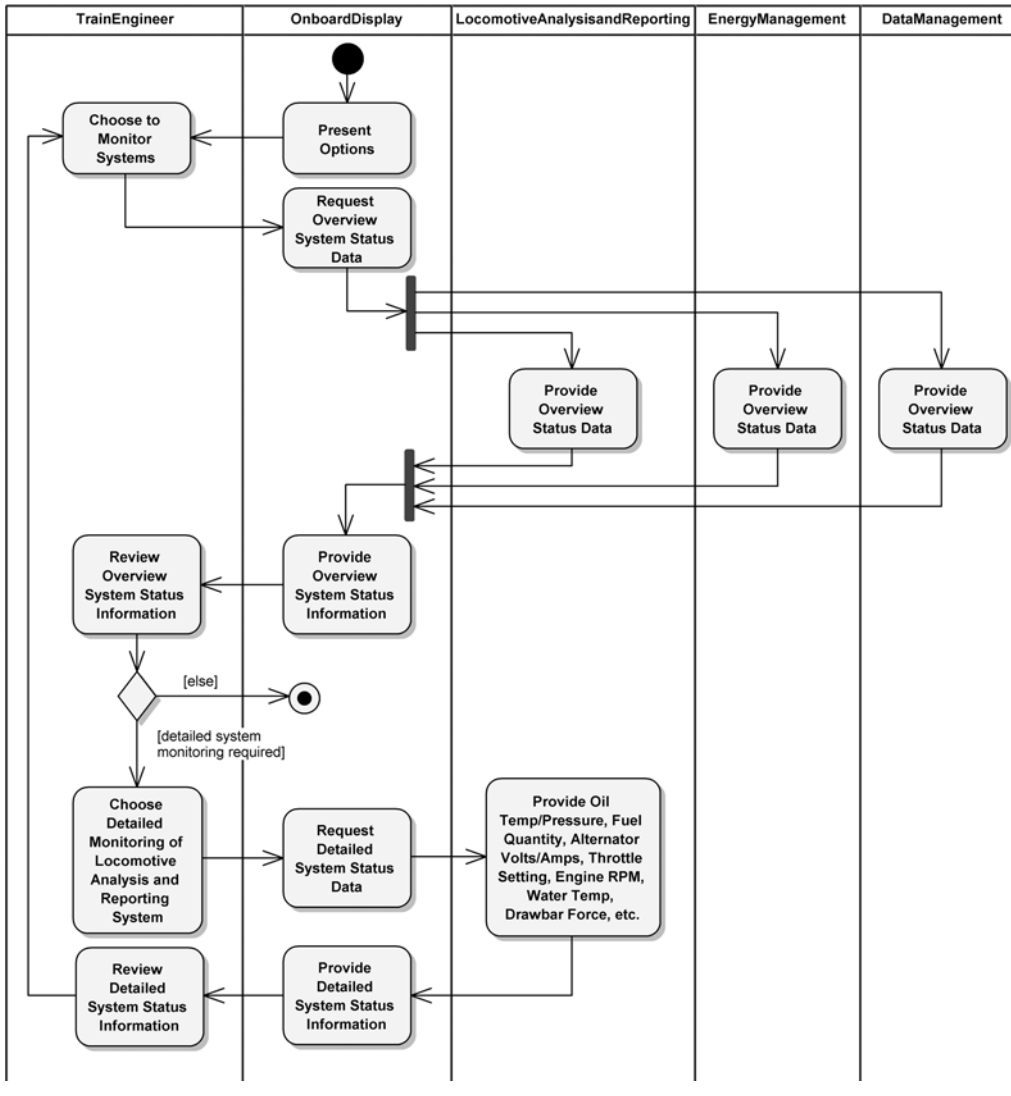

Figure 9–3 A `Monitor Train Systems` Alternate Scenario

It is a matter of project convention whether we regard the activity diagram in Figure 9–3 as representing one (alternate) or two (primary and alternate) separate scenarios. The second alternate scenario we described earlier details the extension of the `Monitor Train Systems` use case functionality with that of the `Predict Failure` use case. This scenario could be appended to Figure 9–3 to provide a more complete picture of system capability by detailing the actions whereby the `TrainEngineer` requests a failure prediction analysis (`condition: {request Predict Failure})` be run on the problematic system. In fact, we show this perspective in the Interaction Overview Diagram sidebar.

Interaction Overview Diagram

Another way to depict the `Monitor Train Systems` use case—with its primary and alternate scenarios—is by using an interaction overview diagram, as shown in Figure 9–4.

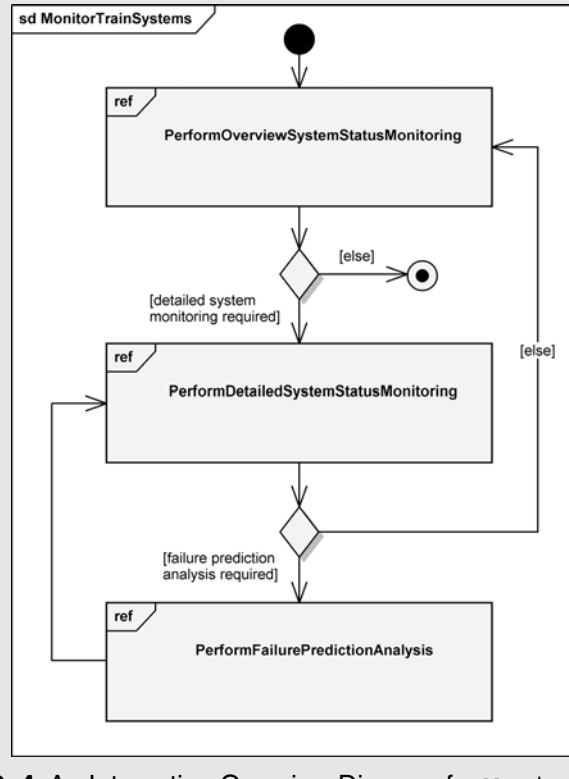

Figure 9–4 An Interaction Overview Diagram for `Monitor Train Systems`

True to its name, this diagram shows a higher-level overview of the complete `Monitor Train Systems` use case functionality. This interaction overview diagram shows a flow among interaction occurrences, indicated by the three frames annotated with `ref` in their upper-left corners. In place of the reference to interaction diagrams, we could show the actual interactions to provide the details for each of the scenarios: `Perform Overview System Status Monitoring`, `Perform Detailed System Status Monitoring`, and `Perform Failure Prediction Analysis`.

The interaction overview diagram can use any type of interaction—sequence, communication, timing, or another interaction overview—to show this detail. As we see here, this diagram can be used to map the flow from one interaction to another, which can be useful if you have long, complicated interactions.

Defining the TTMS Architecture

A much more thorough analysis of the functionality required by all the use case scenarios, including the impact of the nonfunctional requirements and constraints, leads us to a block diagram for the Train Traffic Management System's major elements, as shown in Figure 9–5 [2]. The locomotive analysis and reporting system

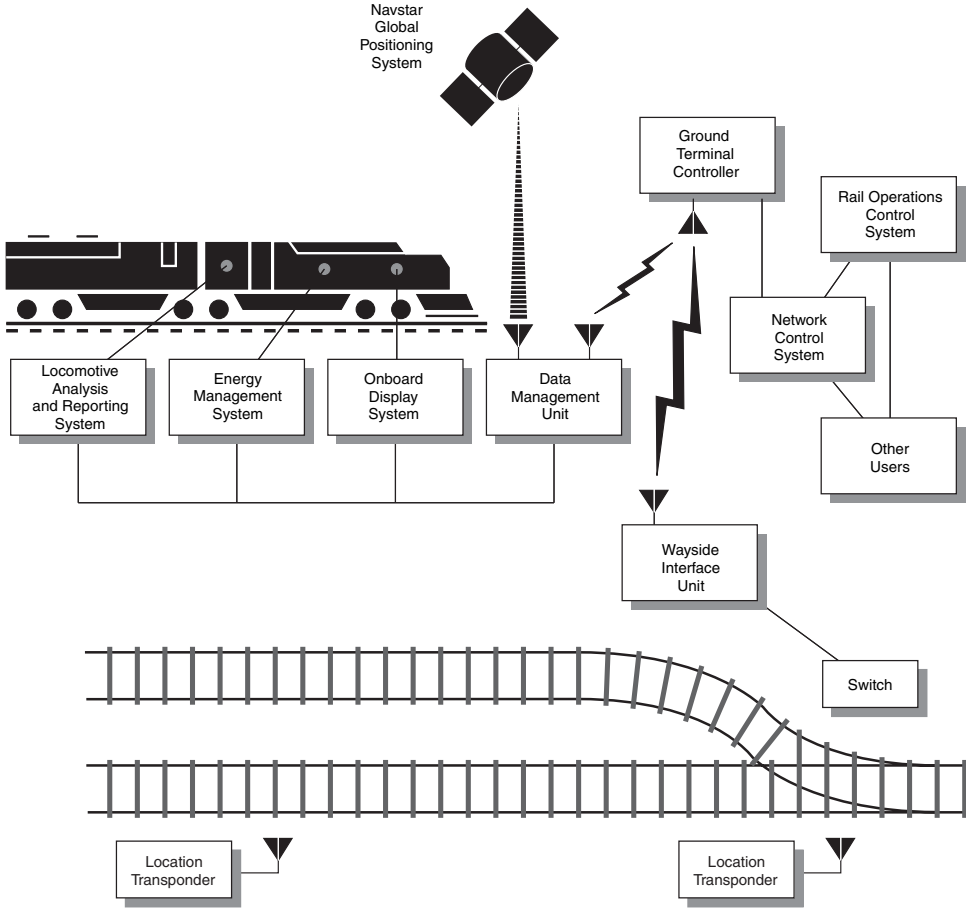

Figure 9–5 The Train Traffic Management System

includes several digital and analog sensors for monitoring locomotive conditions, including oil temperature, oil pressure, fuel quantity, alternator volts and amperes, throttle setting, engine RPM, water temperature, and drawbar force. Sensor values are presented to the train engineer via the onboard display system and to dispatchers and maintainers elsewhere on the network. Warning or alarm conditions are registered whenever certain sensor values fall outside of the normal operating range. A log of sensor values is maintained to support maintenance and fuel management.

The energy management system advises the train engineer in real time as to the most efficient throttle and brake settings. Inputs to this system include track profile and grade, speed limits, schedules, train load, and power available, from which the system can determine fuel-efficient throttle and brake settings that are consistent with the desired schedule and safety concerns. Suggested throttle and brake settings, track profile and grade, and train position and speed are made available for display on the onboard display system.

The onboard display system provides the human/machine interface for the train engineer. Information from the locomotive analysis and reporting system, the energy management system, and the data management unit are made available for display. Soft keys exist to permit the engineer to select different displays.

The data management unit serves as the communications gateway between all onboard systems and the rest of the network, to which all trains, dispatchers, and other users are connected.

Train location tracking is achieved via two devices on the network: location transponders and the Navstar GPS. The locomotive analysis and reporting system can determine the general location of a train via dead reckoning, simply by counting wheel revolutions. This information is augmented by information from location transponders, which are placed every mile along a track and at critical track junctions. These transponders relay their identity to passing trains via their data management units, from which a more exact train location may be determined. Trains may also be equipped with GPS receivers, from which train location may be determined to within 10 yards.

A wayside interface unit is placed wherever there is some controllable device (such as a switch) or a sensor (such as an infrared sensor for detecting overheated wheel bearings). Each wayside interface unit may receive commands from a local ground terminal controller (e.g., to turn a signal on or off). Devices may be overridden by local manual control. Each unit can also report its current setting. A ground terminal controller relays information to and from passing trains and to and from wayside interface units. Ground terminal controllers are placed along a track, spaced close enough so that every train is always within range of at least one terminal.

Every ground terminal controller relays its information to a common network control system. Connections between the network control system and each ground terminal controller may be made via microwave link, landlines, or fiber optics, depending on the remoteness of each ground terminal controller. The network control system monitors the health of the entire network and can automatically route information in alternate ways in the event of equipment failure.

The network control system is ultimately connected to one or more dispatch centers, which comprise the rail operations control system and other users. At the rail operations control system, dispatchers can establish train routes and track the progress of individual trains. Individual dispatchers control different territories; each dispatcher's control console may be set up to control one or more territories. Train routes include instructions for automatically switching trains from track to track, setting speed restrictions, setting out or picking up cars, and allowing or denying train clearance to a specific track section. Dispatchers may note the location of track work along train routes for display to train engineers. Trains may be stopped from the rail operations control system (manually by dispatchers or automatically) when hazardous conditions are detected (such as a runaway train, track failure, or a potential collision condition). Dispatchers may also call up any information available to individual train engineers, as well as send movement authority, wayside device settings, and plan revisions.

It should be apparent that track layouts and wayside equipment may change over time; in addition, the numbers of trains and their routes may change daily. The Train Traffic Management System must therefore be designed to permit incorporation of new sensor, network, and processor technology. Our nonfunctional requirement—to have a designed-in capability to maintain and evolve the TTMS—makes it very clear that we must design an architecture that has the flexibility to evolve over time. In addition, both of our constraints tell us that the system must rely on national standards (government and industry), while maximizing the use of COTS hardware and software.

From Systems Engineering to Hardware and Software Engineering

Up to this point in our development, we performed systems engineering, rather than hardware or software engineering activities, as we analyzed scenarios of the use cases that specify the primary functional requirements for the Train Traffic Management System. From this analysis, we were able to specify a block diagram of its major elements to define a candidate TTMS system architecture. As we continue our development, the architecture of lower-level hardware and software elements will evolve based on the concepts that our system architects likely have in mind. Eventually we decide what portions of the system functionality will

be fulfilled by hardware, software, or manual operations. At that point, development becomes even more of a collaborative effort among the systems, hardware, software, and operational engineering teams.

The block diagram in Figure 9–5 presents a candidate architecture developed using an object-oriented approach, the clear consequence of this being the component nature of the architecture. We see the elements of the TTMS exhibiting cohesion and loose coupling while performing major functions in the system. As we further our analysis of the system's functionality, we may continue to use activity diagrams (especially when working with domain experts), which provide a clear perspective of the work being done by each of the system's elements as they collaborate in the system scenarios. More specific detail of the interactions is required as we proceed through the lower levels of the architectural hierarchy; consequently, we will more likely use sequence diagrams, class diagrams, and prototypes to examine the required system behavior.

In Figure 9–6, we provide a sequence diagram that captures one simple scenario for the automated processing of a daily train order and provides more detail into the inner workings of the Train Traffic Management System than does the activity diagram presented earlier in Figure 9–2. We assume this scenario begins essentially at the conclusion of Figure 9–2, where a new train plan has been created. Here we see just the major events that transpire and the interactions of the system elements. Later in our development, we must begin to document element details such as attribute definitions, operation signatures, and association specifications.

After completing our systems engineering analysis of the TTMS functionality (through its architectural levels), we must allocate the system requirements to hardware, software, and even operational elements. We say "even operational"

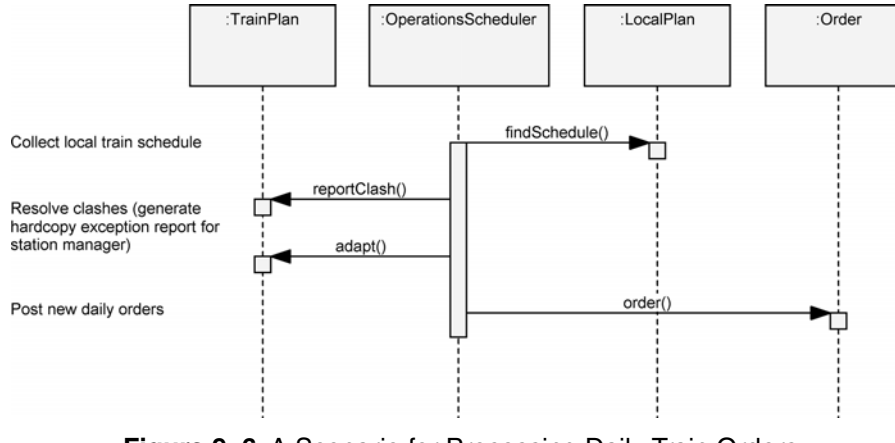

Figure 9–6 A Scenario for Processing Daily Train Orders

because all too often we think that everything can and should be automated. Certainly, that is a typical goal, but we must understand that some functionality, especially in safety-critical areas, should employ (and must by law in some cases) a human-in-the-loop design. In some cases, the allocation of requirements to hardware or software is fairly obvious; for example, software is the right implementation vehicle for describing train schedules. For both the onboard display system and the displays in the rail operations control centers, one might use off-the-shelf terminals or workstations. These allocation decisions are driven by many criteria, including reuse issues, commercially available items, and the experience and preferences of the system architects. When choosing commercially available items, such as the many sensors in the system, we have allocated those element design decisions to the engineers at the vendor companies. In general, though, we will lean toward software where we need the most flexibility and will choose hardware where performance is vital.

For the purposes of our problem, we assume that an initial hardware architecture has been chosen by the system architects. This choice need not be considered irreversible, but at least it gives us a starting point in terms of where to allocate software requirements. As we proceed with analysis and then design, we need the freedom to trade off hardware and software: We might later decide that additional hardware is needed to satisfy some requirement or that certain functions can be performed better through software than hardware.

Figure 9–7 illustrates the target deployment hardware for the Train Traffic Management System, using the notation for deployment diagrams. This hardware architecture parallels the block diagram of the system shown earlier in Figure 9–5. Specifically, there is one computer on each train, encompassing the locomotive analysis and reporting system, the energy management system, the onboard display system, and the data management unit. Each location transponder is connected to a transmitter, through which messages may be sent to passing trains; no computer is associated with a location transponder. On the other hand, each collection of wayside devices (each of which encompasses a wayside interface unit and its switches) is controlled by a wayside computer that may communicate via its transmitter and receiver with a passing train or a ground terminal controller. Communications between transmitters and receivers may be made via microwave link, landlines, or fiber optics, as discussed earlier. Each ground terminal controller ultimately connects to a local area network, one for each dispatch center (encompassing the rail operations control system). Because of the need for uninterrupted service, we have chosen to place two computers at each dispatch center: a primary computer and a backup computer that we expect will be brought online whenever the primary computer fails. During idle periods, the backup computer can be used to service the computational needs of other, lower-priority users.

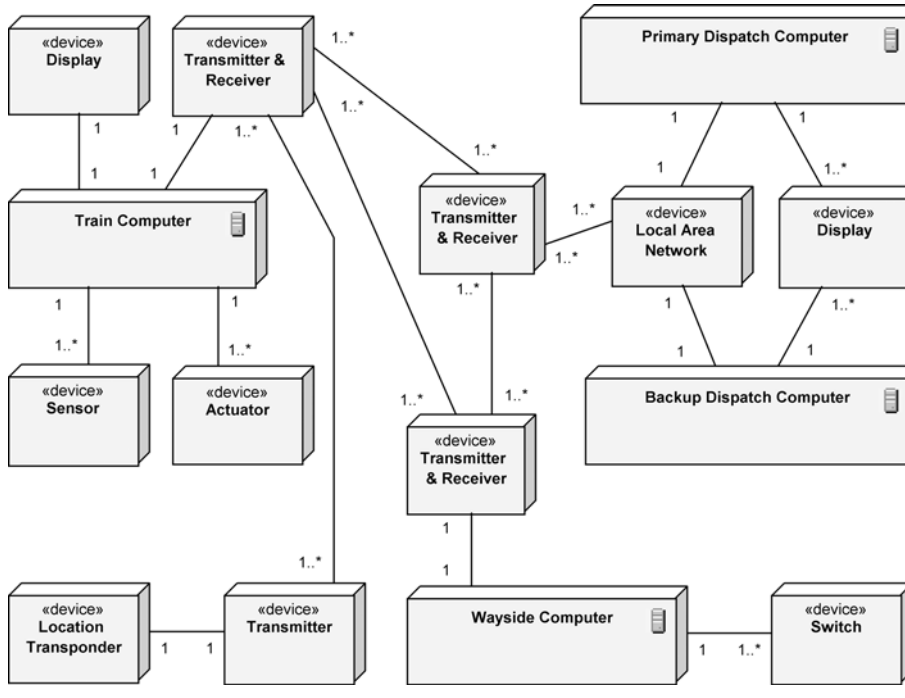

Figure 9–7 The Deployment Diagram for the Train Traffic Management System

When operational, the Train Traffic Management System may involve hundreds of computers, including one for each train, one for each wayside interface unit, and two at each dispatch center. The deployment diagram shows the presence of only a few of these computers since the configurations of similar computers are completely redundant.

The key to maintaining sanity during the development of any complex project is to engineer sound and explicit interfaces among the key elements of the system. This is particularly important when defining hardware and software interfaces. At the start, interfaces can be loosely defined, but they must quickly be formalized so that different parts of the system can be developed, tested, and released in parallel. Well-defined interfaces also make it far easier to make hardware/software trade-offs as opportunities arise, without disrupting already completed parts of the system. Furthermore, we cannot expect all of the developers in a large, possibly globally distributed, development organization to have a complete view and understanding of all parts of the system. We must therefore leave the specification of these key abstractions and mechanisms to our best architects.

Key Abstractions and Mechanisms

A study of the requirements for the Train Traffic Management System suggests that we really have four different subproblems to solve:

1. Networking
2. Database
3. Human/machine interface
4. Real-time analog and digital device control

How did we come to identify these problems as those involving the greatest development risk?

The thread that ties this system together is a distributed communications network. Messages pass by radio from transponders to trains, between trains and ground terminal controllers, between trains and wayside interface units, and between ground terminal controllers and wayside interface units. Messages must also pass between dispatch centers and individual ground terminal controllers. The safe operation of this entire system depends on the timely and reliable transmission and reception of messages.

Additionally, this system must keep track of the current locations and planned routes of many different trains simultaneously. We must keep this information current and self-consistent, even in the presence of concurrent updates and queries from around the network. This is basically a distributed database problem.

The engineering of the human/machine interfaces poses a different set of problems. Specifically, the users of this system are principally train engineers and dispatchers, none of whom are necessarily skilled in using computers. The user interface of an operating system such as UNIX or Windows might be acceptable to a professional software engineer, but it is often regarded as user-hostile by end users of applications such as the Train Traffic Management System. All forms of user interaction must therefore be carefully engineered to suit this domain-specific group of users.

Lastly, the Train Traffic Management System must interact with a variety of sensors and actuators. No matter what the device, the problems of sensing and controlling the environment are similar and so should be dealt with in a consistent manner by the system.

Each of these four subproblems involves largely independent issues. Our system architects need to identify the key abstractions and mechanisms involved in each, so that we can assign experts in each domain to tackle their particular subproblem in parallel with the others. Note that this is not a problem of analysis or design:

Our analysis of each problem will impact our architecture, and our designs will uncover new aspects of the problem that require further analysis. Development is thus unavoidably iterative and incremental.

If we do a brief domain analysis across these four subproblem areas, we find that there are three common high-level key abstractions:

1. Trains: including locomotives and cars
2. Tracks: encompassing profile, grade, and wayside devices
3. Plans: including schedules, orders, clearances, authority, and crew assignments

Every train has a current location on the tracks, and each train has exactly one active plan. Similarly, the number of trains at each point on the tracks may be zero or one; for each plan, there is exactly one train, involving many points on the tracks.

Continuing, we may devise a key mechanism for each of the four nearly independent subproblems:

1. Message passing
2. Train schedule planning
3. Displaying information
4. Sensor data acquisition

These four mechanisms form the soul of our system. They represent approaches to what we have identified as the areas of highest development risk. It is therefore essential that we deploy our best system architects here to experiment with alternative approaches and eventually settle on a framework from which more junior developers may compose the rest of the system.

9.3 Construction

Architectural design involves the establishment of the central class structure of the system, plus a specification of the common collaborations that animate these classes. Focusing on these mechanisms early directly attacks the elements of highest risk in the system and concretely captures the vision of the system's architects. Ultimately, the products of this phase serve as the framework of classes and collaborations on which the other functional elements of the final system build.

In this section, we start by examining the semantics of each of this system's four key mechanisms: message passing, train schedule planning, displaying informa-

tion, and sensor data acquisition. This leads into a discussion of release management, which supports our iterative and incremental development process. We conclude this section by analyzing how developing a system architecture supports the specification of the TTMS subsystems.

Message Passing

By *message*, we do not mean to imply method invocation, as in an object-oriented programming language; rather, we are referring to a concept in the vocabulary of the problem domain, at a much higher level of abstraction. For example, typical messages in the Train Traffic Management System include signals to activate wayside devices, indications of trains passing specific locations, and orders from dispatchers to train engineers. In general, these kinds of messages are passed at two different levels within the TTMS:

1. Between computers and devices
2. Among computers

Our interest is in the second level of message passing. Because our problem involves a geographically distributed communications network, we must consider issues such as noise, equipment failure, and security.

We can make a first cut at identifying these messages by examining each pair of communicating computers, as shown in our previous deployment diagram (refer back to Figure 9–7). For each pair, we must ask three questions.

1. What information does each computer manage?
2. What information should be passed from one computer to the other?
3. At what level of abstraction should this information be?

There is no determinate solution for these questions. Rather, we must use an iterative approach until we are satisfied that the right messages have been defined and that there are no communications bottlenecks in the system (perhaps because of too many messages over one path, or messages being too large or too small).

It is absolutely critical at this level of design to focus on the substance, not the form, of these messages. Too often, we have seen system architects start off by selecting a bit-level representation for messages. The real problem with prematurely choosing such a low-level representation is that it is guaranteed to change and thus disrupt every client that depends on a particular representation. Furthermore, at this point in the design process, we cannot know enough about how these messages will be used to make intelligent decisions about time- and space-efficient representations.

By focusing on the substance of these messages, we mean to urge a focus on the outside view of each class of messages. In other words, we must decide on the roles and responsibilities of each message and what operations we can meaningfully perform on each message.

The class diagram in Figure 9–8 captures our design decisions regarding some of the most important messages in the Train Traffic Management System. Note that all messages are ultimately instances of a generalized abstract class named `Message`, which encompasses the behavior common to all messages. Three lower-level classes represent the major categories of messages, namely, `TrainStatusMessage`, `TrainPlanMessage`, and `WaysideDeviceMessage`. Each of these classes is further specialized. Indeed, our final design might include dozens of such specialized classes, at which time the existence of these intermediate classes becomes even more important; without them, we would end up with many unrelated—and therefore difficult to maintain—components representing each distinct specialized class. As our design unfolds, we are likely to discover other important groupings of messages and so invent other intermediate classes. Fortunately, reorganizing our class hierarchy in this manner tends to have minimal semantic impact on the clients that ultimately use the base classes.

As part of the architectural design, we would be wise to stabilize the interface of the key message classes early. We might start with a domain analysis of the more interesting base classes in this hierarchy, in order to formulate the roles and responsibilities of all such classes.

Once we have designed the interface of the more important messages, we can write programs that build on these classes to simulate the creation and reception of streams of messages. We can use these programs as a temporary scaffolding to test different parts of the system during development and before the pieces with which they interface are completed.

The class diagram in Figure 9–8 is unquestionably incomplete. In practice, we find that we can identify the most important messages first and let all others evolve as we uncover the less common forms of communication. Using an object-oriented architecture allows us to add these messages incrementally without disrupting the existing design of the system because such changes are generally upwardly compatible.

Once we are satisfied with this class structure, we can begin to design the message-passing mechanism itself. Here we have two competing goals for the mechanism: It must provide for the reliable delivery of messages and yet do so at a high enough level of abstraction so that clients need not worry about how message delivery takes place. Such a message-passing mechanism allows its clients to make simplifying assumptions about how messages are sent and received.

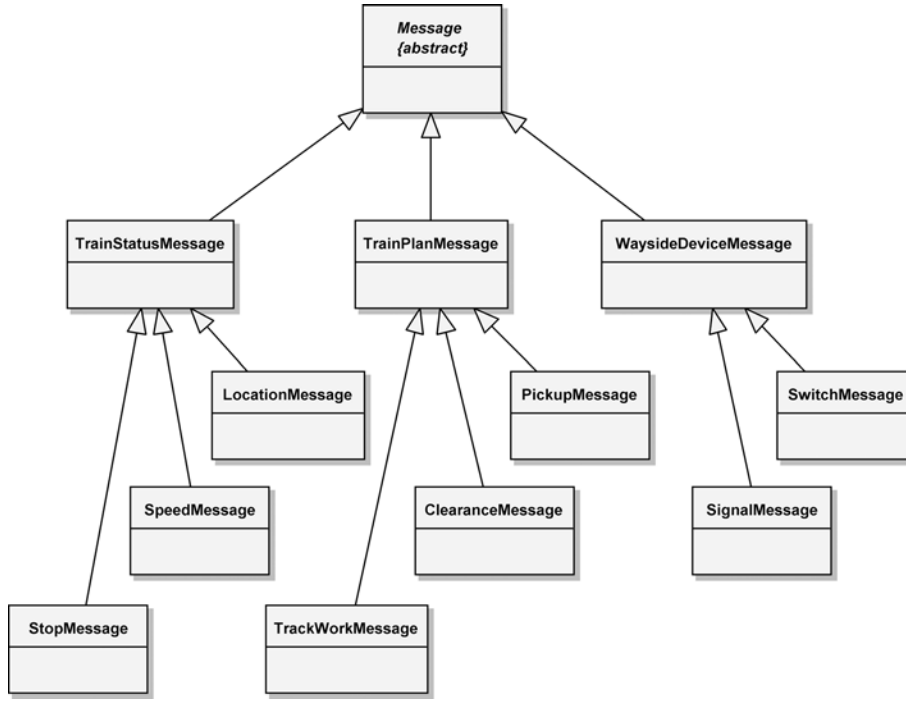

Figure 9–8 The `Message` Class Diagram

Figure 9–9 provides a scenario that captures our design of the message-passing mechanism. As this diagram indicates, to send a message, a client first creates a new message `m` and then broadcasts it to its node's message manager, whose responsibility is to queue the message for eventual transmission. Notice that our design uses four objects that are active (have their own thread of control), as indicated by the extra vertical lines within the object notation: `Client`, `messageMgr :Queue`, `messageMgr' :Queue`, and `Receiver`. Notice also that the message manager receives the message to be broadcast as a parameter and then uses the services of a `Transporter` object to reduce the message to its canonical form and broadcast it across the network.

As this diagram suggests, we choose to make this an asynchronous operation—indicated by the open-headed arrow—because we don't want to make the client wait for the message to be sent across a radio link, which requires time for encoding, decoding, and perhaps retransmission because of noise. Eventually, some `Listener` object on the other side of the network receives this message and presents it in a canonical form to its node's message manager, which in turn creates a parallel message and queues it. A receiver can block at the head of its message

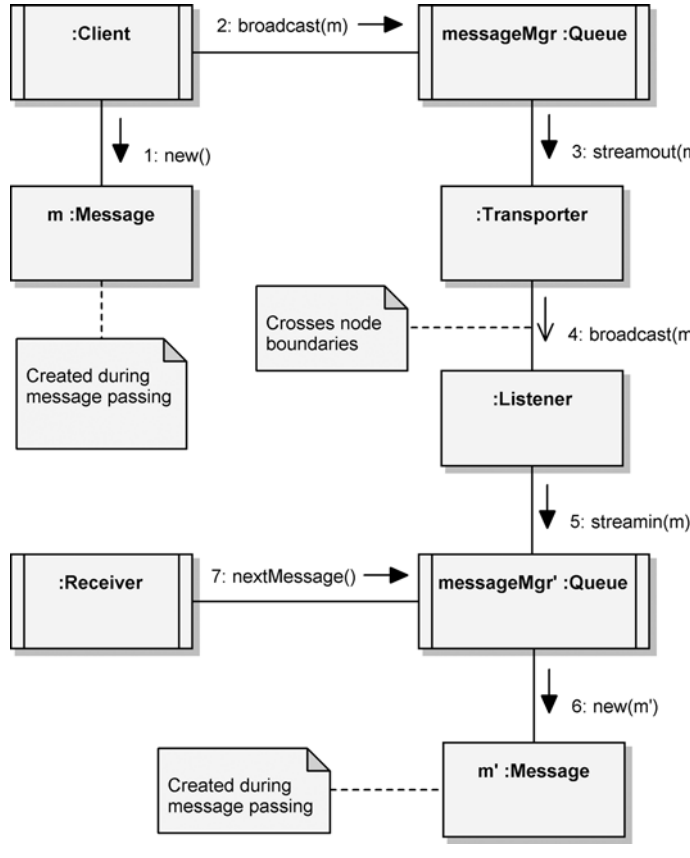

Figure 9–9 The Message-Passing Mechanism

manager's queue, waiting for the next message to arrive, which is delivered as a parameter to the operation `nextMessage()`, a synchronous operation.

Our design of the message manager places it at the application layer in the ISO Open Systems Interconnection (OSI) model for networks [3]. This allows all message-sending clients and message-receiving clients to operate at the highest level of abstraction, namely, in terms of application-specific messages.

We expect the final implementation of this mechanism to be a bit more complex. For example, we might want to add behaviors for encryption and decryption and introduce codes to detect and correct errors, so as to ensure reliable communication in the presence of noise or equipment failures.

Train Schedule Planning

As we noted earlier, the concept of a train plan is central to the operation of the Train Traffic Management System. Each train has exactly one active plan, and each plan is assigned to exactly one train and may involve many different orders and locations on the track.

Our first step is to decide exactly what parts constitute a train plan. To do so, we need to consider all the potential clients of a plan and how we expect each of them to use that plan. For example, some clients might be allowed to create plans, others might be allowed to modify plans, and still others might be allowed only to read plans. In this sense, a train plan acts as a repository for all the pertinent information associated with the route of one particular train and the actions that take place along the way, such as picking up or setting out cars.

Figure 9–10 captures our strategic decisions regarding the structure of the `TrainPlan` class. We use a class diagram to show the parts that compose a train plan (much as a traditional entity-relationship diagram would do). Thus, we see that each train plan has exactly one crew and may have many general orders and

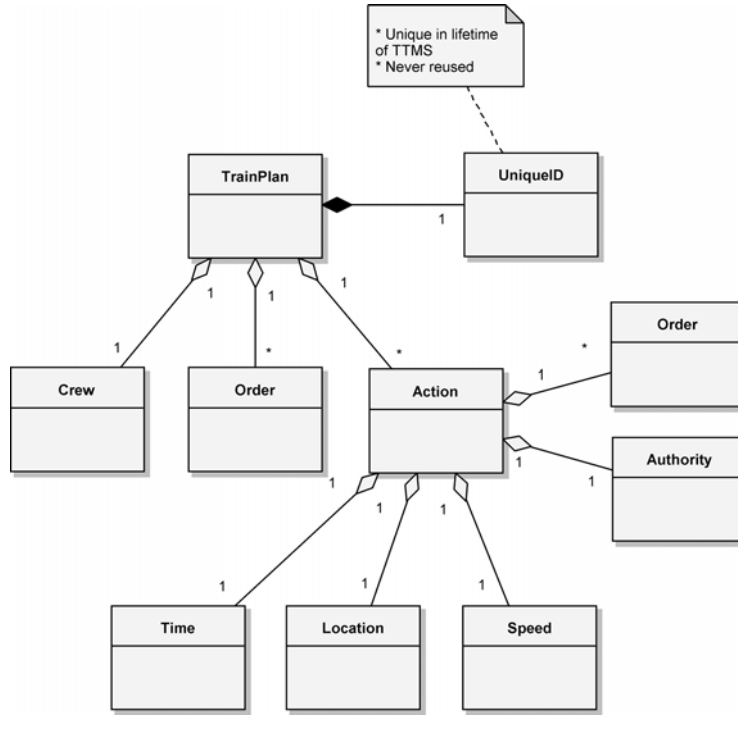

Figure 9–10 The `TrainPlan` Class Diagram

many actions. We expect these actions to be time ordered, with each action composed of information such as time, a location, speed, authority, and orders. For example, a specific train plan might consist of the actions shown in Table 9–1.

As the diagram in Figure 9–10 indicates, the `TrainPlan` class has a `UniqueId`, whose purpose is to provide a number for uniquely identifying each `TrainPlan` instance. Because of the complexity of the information here, classes in this diagram that might otherwise be considered an attribute of a class are really stand-alone classes. For example, the `UniqueID` class is not merely an identification number; it contains various attributes and operations necessary to meet stringent national and international regulations. Another example is that crews have restrictions placed on their work—they may work only at certain locations or must adhere to speed restrictions in certain locations at particular times.

As we did for the `Message` class and its subclasses, we can design the most important elements of a train plan early in the development process; its details will evolve over time, as we actually apply plans to various kinds of clients.

The fact that we may have a plethora of active and inactive train plans at any one time confronts us with the database problem we spoke of earlier. The class diagram in Figure 9–10 can serve as an outline for the logical schema of this database. The next question we might therefore ask is simply, where are train plans kept?

In a more perfect world, with no communication noise or delays and infinite computing resources, our solution would be to place all train plans in a single, centralized database. This approach would yield exactly one instance of each train plan. However, the real world is much more perverse, so this solution is not practical. We must expect communication delays, and we don't have unlimited processor cycles. Thus, having to access a plan located in the dispatch center from a train would not at all satisfy our real-time and near real-time requirements.

However, we can create the illusion of a single, centralized database in our software. Basically, our solution is to have a database of train plans located on the

Table 9–1 Actions a Train Plan Might Contain

Time	Location	Speed	Authority	Orders
0800	Pueblo	As posted	See yardmaster	Depart yard
1100	Colorado Springs	40 mph		Set out 30 cars
1300	Denver	45 mph		Set out 20 cars
1600	Pueblo	As posted		Return to yard

computers at the dispatch center, with copies of individual plans distributed as needed at sites around the network. For efficiency, then, each train computer could retain a copy of its current plan. Thus, onboard software could query this plan with negligible delay. If the plan changed, either as a result of dispatcher action or (less likely) by the decision of the train engineer, our software would have to ensure that all copies of that plan were updated in a timely fashion.

The way this scenario plays out is a function of our train schedule planning mechanism, shown in Figure 9–11. The primary version of each train plan resides in a centralized database at a dispatch center, with zero or more mirror-image copies scattered about the network. Whenever some client requests a copy of a particular train plan (via the operation `get()`, invoked with a value of `UniqueId` as an argument), the primary version is cloned and delivered to the client as a parameter, and the network location of the copy is recorded in the database. Now, suppose that a client on a train needed to make a change to a particular plan, perhaps as a result of some action by the train engineer. Ultimately, this client would invoke operations on its copy of the train plan and so modify its state. These operations would also send messages to the centralized database, to modify the state of the primary version of the plan in the same way. Since we record the location in the network of each copy of a train plan, we can also broadcast messages to the centralized repository that force a corresponding update to the state of all remaining copies. To ensure that changes are made consistently across the network, we

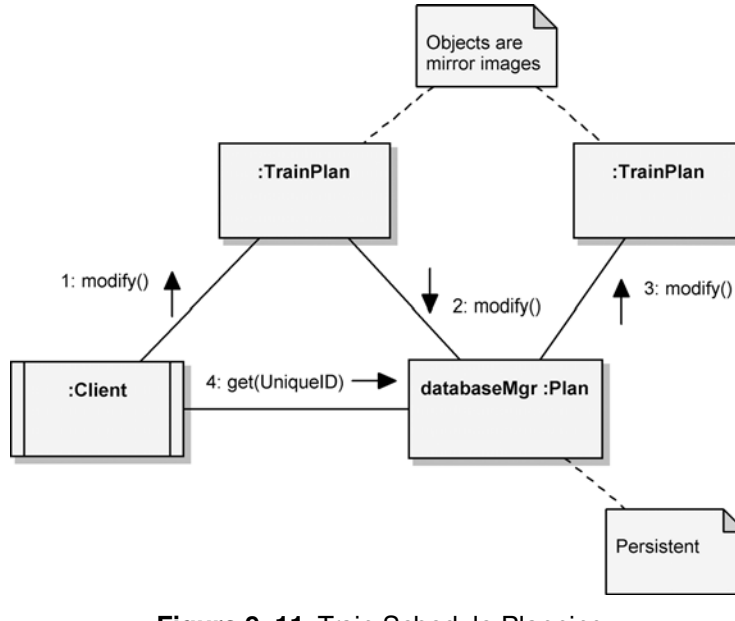

Figure 9–11 Train Schedule Planning

could employ a record-locking mechanism, so that changes would not be committed until all copies and the primary version were updated.

This mechanism applies equally well if some client at the dispatch center initiates the change, perhaps as a result of some dispatcher action. First, the primary version of the plan would be updated, and then changes to all copies would be broadcast throughout the network, using the same mechanism. In either case, how exactly do we broadcast these changes? The answer is that we use the message-passing mechanism devised earlier. Specifically, we would need to add to our design some new train plan messages and then build our train plan mechanism on this lower-level message-passing mechanism.

Using commercial, off-the-shelf database management systems on the dispatch computers allows us to address any requirements for database backup, recovery, audit trails, and security.

Displaying Information

Using off-the-shelf technology for our database needs helps us to focus on the domain-specific parts of our problem. We can achieve similar leverage for our display needs by using standard graphics facilities. Using off-the-shelf graphics software effectively raises the level of abstraction in our system, so that developers never need to worry about manipulating the visual representation of displayable objects at the pixel level. Still, it is important to encapsulate our design decisions regarding how various objects are represented visually.

For example, consider displaying the profile and grade of a specific section of track. Our requirements dictate that such a display may appear in two different places: at a dispatch center and onboard a train (with the display focusing only on the track that lies ahead of the train). Assuming that we have some class whose instances represent sections of track, we might take two approaches to representing the state of such objects visually. First, we might have some display manager object that builds a visual representation by querying the state of the object to be displayed. Alternately, we could eliminate this external object and have each displayable object encapsulate the knowledge of how to display itself. We prefer this second approach because it is simpler and more in the spirit of the object model.

There is a potential disadvantage to this approach, however. Ultimately, we might have many different kinds of displayable objects, each implemented by different groups of developers. If we let the implementation of each displayable object proceed independently, we are likely to end up with redundant code, different implementation styles, and a generally unmaintainable mess. A far better solution is to do a domain analysis of all the kinds of displayable objects, determine what visual elements they have in common, and devise an intermediate set of class util-

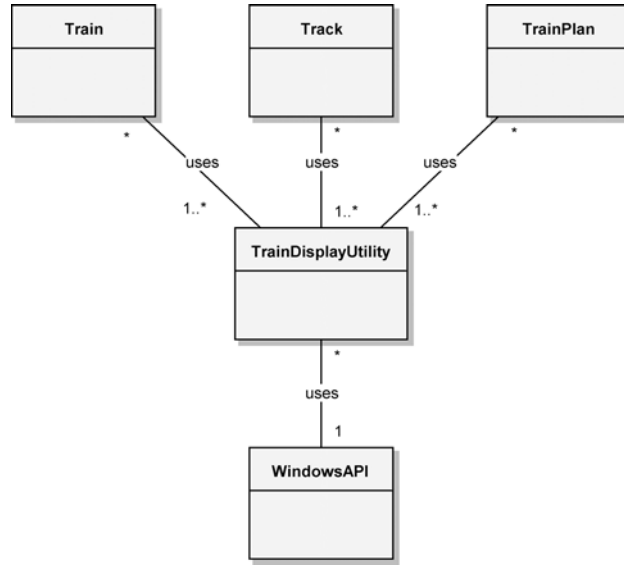

Figure 9–12 Class Utilities for Displaying

ities that provide display routines for these common picture elements. These class utilities in turn can build on lower-level, off-the-shelf graphics packages.

Figure 9–12 illustrates this design, showing that the implementation of all displayable objects shares common class utilities. These utilities in turn build on lower-level Windows interfaces, which are hidden from all of the higher-level classes. Pragmatically, interfaces such as the Windows API cannot easily be expressed in a single class. Therefore, our diagram is a bit of a simplification: It is more likely that our implementation will require a set of peer class utilities for the Windows API as well as for the train display utilities.

The principal advantage of this approach is that it limits the impact of any lower-level changes resulting from hardware/software trade-offs. For example, if we find that we need to replace our display hardware with more or less powerful devices, we need only reimplement the routines in the `TrainDisplayUtility` class. Without this collection of routines, low-level changes would require us to modify the implementation of every displayable object.

Sensor Data Acquisition

As our requirements suggest, the Train Traffic Management System includes many different kinds of sensors. For example, sensors on each train monitor the oil temperature, fuel quantity, throttle setting, water temperature, drawbar load,

and so on. Similarly, active sensors in some of the wayside devices report among other things the current positions of switches and signals. The kinds of values returned by the various sensors are all different, but the processing of different sensor data is all very much the same. Furthermore, most sensors must be sampled periodically. If a value is within a certain range, nothing special happens other than notifying some client of the new value. If this value exceeds certain preset limits, a different client might be warned. Finally, if this value goes far beyond its limits, we might need to sound some sort of alarm and notify yet another client to take drastic action (e.g., when locomotive oil pressure drops to dangerous levels).

Replicating this behavior for every kind of sensor not only is tedious and error-prone but also usually results in redundant code. Unless we exploit this commonality, different developers will end up inventing multiple solutions to the same problem, leading to the proliferation of slightly different sensor mechanisms and, in turn, a system that is more difficult to maintain. It is highly desirable, therefore, to do a domain analysis of all periodic, nondiscrete sensors, so that we might invent a common sensor mechanism for all kinds of sensors. We might use an architecture that encompasses a hierarchy of sensor classes and a frame-based mechanism that periodically acquires data from these sensors.

Release Management

Since we are using an incremental development approach, we will investigate the employment of release management techniques and further analyze the system's architecture and the specification of its subsystems.

We start the incremental development process by first selecting a small number of interesting scenarios, taking a vertical slice through our architecture, and then implementing enough of the system to produce an executable product that at least simulates the execution of these scenarios.

For example, we might select just the primary scenarios of three use cases: `Route Train`, `Monitor Train Systems`, and `Monitor Traffic`. Together, the implementation of these three scenarios requires us to touch almost every critical architectural interface, thereby forcing us to validate our strategic assumptions. Once we successfully pass this milestone, we might then generate a stream of new releases, according to the following sequence.

1. Create a train plan based on an existing one; modify a train plan.
2. Request detailed monitoring of a system with a yellow condition; request a failure prediction analysis; request maintainer review of a failure prediction analysis.

3. Manually avoid a collision; request automated assistance in avoiding a collision; track train traffic using either TTMS resources or Navstar GPS.

For a 12- to 18-month development cycle, this probably means generating a reasonably stable release every 3 months or so, each building on the functionality of the other. When we are done, we will have covered every scenario in the system.

The key to success in this strategy is risk management, whereby for each release we identify the highest development risks and attack them directly. For control system applications such as this one, this means introducing testing of positive control early (so that we identify any system control holes early enough that we can do something about them). As our sequence of releases suggests, this also means broadly selecting scenarios for each release from across the functional elements of the system, so that we are not blindsided by unforeseen gaps in our analysis.

System Architecture

The software design for very large systems must often commence before the target hardware is completed. Software design frequently takes far longer than hardware design, and in any case, trade-offs must be made against each along the way. This implies that hardware dependencies in the software must be isolated to the greatest extent possible, so that software design can proceed in the absence of a stable target environment. It also implies that the software must be designed with the idea of replaceable subsystems in mind. In a command and control system such as the Train Traffic Management System, we might wish to take advantage of new hardware technology that has matured during the development of the system's software.

We must also have an early and intelligent physical decomposition of the system's software, so that subcontractors working on different parts of the system can work in parallel. Often many nontechnical reasons drive the physical decomposition of a large system. Perhaps the most important of these concerns the assignment of work to independent teams of developers. Subcontractor relationships are usually established early in the life of a complex system, often before there is enough information to make sound technical decisions regarding proper subsystem decomposition.

How do we select a suitable subsystem decomposition? The highest-level objects are often clustered around functional lines. Again, this is not orthogonal to the object model because by the term *functional*, we do not mean algorithmic abstractions, embodying simple input/output mappings. We are speaking of scenarios that represent outwardly visible and testable behaviors, resulting from the cooperative action of logical collections of objects. Thus, the highest-level abstractions

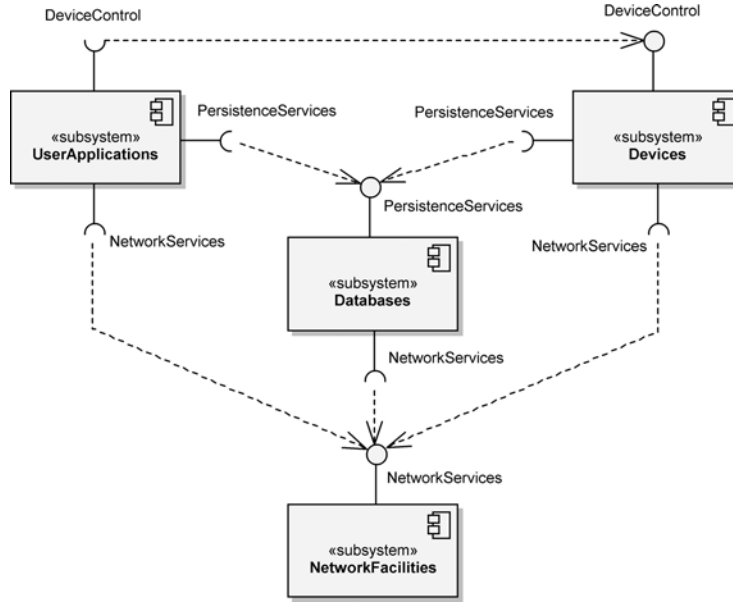

Figure 9–13 The Top-Level Component Diagram for the Train Traffic Management System

and mechanisms that we first identify are good candidates around which to organize our subsystems. We may assert the existence of such subsystems first and then evolve their interfaces over time.

The component diagram shown in Figure 9–13 represents our design decisions regarding the top-level system architecture of the Train Traffic Management System. Here we see a layered architecture that encompasses the functions of the four subproblems we identified earlier, namely, networking, database, the human/machine interface, and real-time device control.

Subsystem Specification

If we focus on the outside view of any of these subsystems, we find that it has all the characteristics of an object. It has a unique, albeit static, identity; it embodies a significant amount of state; and it exhibits very complex behavior. Subsystems serve as the repositories of other subsystems and eventually classes; thus, they are best characterized by the resources they export through their provided interfaces, such as the `NetworkServices` provided by the `NetworkFacilities` subsystem shown in Figure 9–13.

The component diagram in Figure 9–13 is merely a starting point for the specification of the TTMS subsystem architecture. These top-level subsystems must be further decomposed through multiple architectural levels of nested subsystems. Looking at the `NetworkFacilities` subsystem, we decompose it into two other subsystems, a private `RadioCommunication` subsystem and a public `Messages` subsystem. The private subsystem hides the details of software control of the physical radio devices, while the public subsystem provides the functionality of the message-passing mechanism we designed earlier.

The subsystem named `Databases` builds on the resources of the subsystem `NetworkFacilities` and implements the train plan mechanism we created earlier. We choose to further decompose this subsystem into two public subsystems, representing the major database elements in the system. We name these nested subsystems `TrainPlanDatabase` and `TrackDatabase`, respectively. We also expect to have one private subsystem, `DatabaseManager`, whose purpose is to provide all the services common to the two domain-specific databases.

In the `Devices` subsystem, we choose to group the software related to all wayside devices into one subsystem and the software associated with all onboard locomotive actuators and sensors into another. These two subsystems are available to clients of the `Devices` subsystem, and both are built on the resources of the `TrainPlanDatabase` and `Messages` subsystems. Thus, we have designed the `Devices` subsystem to implement the sensor mechanism we described earlier.

Finally, we choose to decompose the top level `UserApplications` subsystem into several smaller ones, including the subsystems `EngineerApplications` and `DispatcherApplications`, to reflect the different roles of the two main users of the Train Traffic Management System. The subsystem `Engineer Applications` includes resources that provide all the train-engineer/machine interaction specified in the requirements, including the functionality of the locomotive analysis and reporting system and the energy management system. We include the subsystem `DispatcherApplications` to encompass the software that provides the functionality of all dispatcher/machine interactions. Both `EngineerApplications` and `DispatcherApplications` share common private resources, as provided from the subsystem `Displays`, which embodies the display mechanism we described earlier.

This design leaves us with four top-level subsystems, encompassing several smaller ones, to which we have allocated all of the key abstractions and mechanisms we invented earlier. These subsystems are allocated to development teams that will design and implement them, maintaining adherence to the defined interfaces through which each subsystem will collaborate with other subsystems at the same level of abstraction.

This approach to decomposing a large and complex problem affords us different views of the system while it is being developed. A system release is thus composed of compatible versions of each subsystem, and we may have many such releases: one for each developer, one for our quality assurance team, and perhaps one for early customer use. Individual developers can create their own stable releases into which they integrate new versions of the software for which they are responsible, before releasing it to the rest of the team.

The key to making this work is the careful engineering of subsystem interfaces. Once engineered, these interfaces must be rigorously guarded. How do we determine the outside view of each subsystem? We do so by looking at each subsystem as an object. Thus, we ask the same questions we ask of much more primitive objects: What state does this object embody, what operations can clients meaningfully perform on it, and what operations does it require of other objects?

For example, consider the subsystem `TrainPlanDatabase`. It builds on three other subsystems (`Messages`, `TrainDatabase`, and `TrackDatabase`) and has several important clients, namely, the four subsystems, `WaysideDevices`, `LocomotiveDevices`, `EngineerApplications`, and `DispatcherApplications`. The `TrainPlanDatabase` embodies a relatively straightforward state, specifically, the state of all train plans. Of course, the twist is that this subsystem must support the behavior of the distributed train plan mechanisms. Thus, from the outside, clients see a monolithic database, but from the inside, we know that this database is really distributed and must therefore be constructed on top of the message-passing mechanism found in the subsystem `Messages`.

What services does the `TrainPlanDatabase` provide? All the usual database operations seem to apply: adding records, deleting records, modifying records, and querying records. We would eventually capture all of these design decisions that make up this subsystem in the form of classes that provide the declarations of all these operations.

At this stage in the design, we would continue the design process for each subsystem. Again, we expect that these interfaces will not be exactly right at first; we must allow them to evolve over time. Happily, as for smaller objects, our experience suggests that most of the changes we will need to make to these interfaces will be upwardly compatible, assuming that we did a good job up front in characterizing the behavior of each subsystem in an object-oriented manner.

9.4 Post-Transition

A safe and useful control system is a work in progress. This is not to say that we never get to the point where we have a stable system—in fact, we must with every delivery. Rather, the reality is that for systems that are central to a business such as rail transportation, the hardware and software must adapt as the rules of the business change; otherwise, our system becomes a liability rather than a competitive asset. Though the dominant risk in changes to a system like the Train Traffic Management System is technical, there are political and social risks as well. By having a resilient object-oriented architecture, the development organization at least offers the company many degrees of freedom in being able to adapt nimbly to the changing regulatory environment and marketplace.

For the Train Traffic Management System, we can envision a significant addition to our requirements, namely, payroll processing. Specifically, suppose that our analysis shows that train company payroll is currently being supported by a piece of hardware that is no longer being manufactured and that we are at great risk of losing our payroll processing capability because a single serious hardware failure would put our accounting system out of action forever. For this reason, we might choose to integrate payroll processing with the Train Traffic Management System. At first, it is not difficult to conceive how these two seemingly unrelated problems could coexist; we could simply view them as separate applications, with payroll processing running as a background activity.

Further examination shows that there is actually tremendous value to be gained from integrating payroll processing. You may recall from our earlier discussion that, among other things, train plans contain information about crew assignments. Thus, it is possible for us to track actual versus planned crew assignments, and from this we can calculate hours worked, amount of overtime, and so on. By getting this information directly, our payroll calculations will be more precise and certainly timelier.

What does adding this functionality do to our existing design? Very little. Our approach would be to add one more subsystem, representing the functionality of payroll processing, inside the `UserApplications` subsystem. At this location in the architecture, such a subsystem would have visibility to all the important mechanisms on which it could build. This is indeed quite common in well-structured object-oriented systems: A significant addition in the requirements for the system can be dealt with fairly easily by building new applications on existing mechanisms.

An even more significant change would be to inject expert system technology into our system by building a dispatcher's assistant that could provide advice about

appropriate traffic routing and emergency responses. What would be the impact to the system's architecture?

Again, there would be very little impact. Our solution would be to add a new subsystem between the subsystems `TrainPlanDatabase` and `DispatcherApplications` because the knowledge base embodied by this expert system parallels the contents of the `TrainPlanDatabase`; furthermore, the subsystem `DispatcherApplications` is the sole client of this expert system. We would need to invent some new mechanisms to establish the manner in which advice is presented to the ultimate user. For example, we might use a blackboard architecture.

One fascinating characteristic of architectures is that—if well engineered—they tend to reach a sort of critical mass of functionality and adaptability. In other words, if we have selected the right element functionality and structure, we will find that users soon discover means to evolve the system functionality in ways its designers never imagined or expected. As we discover patterns in the ways that clients use our system, it makes sense to codify these patterns by formally making them a part of the architecture. A sign of a well-designed architecture is that we can introduce these new patterns by reusing existing mechanisms and thus preserving its design integrity.

Artificial Intelligence: Cryptanalysis

Sentient creatures exhibit a vastly complex set of behaviors that spring from the mind through mechanisms that we only poorly understand. For example, think about how you solve the problem of planning a route through a city to run a set of errands. Consider also how, when walking through a dimly lit room, you are able to recognize the boundaries of objects and avoid stumbling. Furthermore, think about how you can focus on one conversation at a party while dozens of people are talking simultaneously. None of these kinds of problems lends itself to a straightforward algorithmic solution. Optimal route planning is known to be an NP-complete problem. Navigating through dark terrain involves deriving understanding from visual input that is (very literally) fuzzy and incomplete. Identifying a single speaker from dozens of sources requires that the listener distinguish meaningful data from noise and then filter out all unwanted conversations from the remaining cacophony.

Researchers in the field of artificial intelligence have pursued these and similar problems to improve our understanding of human cognitive processes. Activity in this field often involves the construction of intelligent systems that mimic certain aspects of human behavior. Erman, Lark, and Hayes-Roth point out that:

> intelligent systems differ from conventional systems by a number of attributes, not all of which are always present:
>
> - They pursue goals which vary over time.
> - They incorporate, use, and maintain knowledge.
> - They exploit diverse, ad hoc subsystems embodying a variety of selected methods.

- They interact intelligently with users and other systems.
- They allocate their own resources and attention. [1]

Any one of these properties is sufficiently demanding to make the crafting of intelligent systems a very difficult task. When we consider that intelligent systems are being developed for a variety of domains that affect both life and property, such as for medical diagnosis or aircraft routing, the task becomes even more demanding because we must design these systems so that they are never actively dangerous: Artificial intelligences rarely embody any kind of commonsense knowledge.

Although the field has at times been oversold by an overly enthusiastic press, the study of artificial intelligence has given us some very sound and practical ideas, among which we count approaches to knowledge representation and the evolution of common problem-solving architectures for intelligent systems, including rule-based expert systems and the blackboard model [2]. In this chapter, we turn to the design of an intelligent system that solves cryptograms using a blackboard framework in a manner that parallels the way a human would solve the same problem. As we will see, the use of object-oriented development is very well suited to this domain.

10.1 Inception

Our problem is one of cryptanalysis, the process of transforming ciphertext back to plaintext. In its most general form, deciphering cryptograms is an intractable problem that defies even the most sophisticated techniques. Happily, our problem is relatively simple because we limit ourselves to single substitution ciphers.

Cryptanalysis Requirements

Cryptography "embraces methods for rendering data unintelligible to unauthorized parties" [3]. Using cryptographic algorithms, messages (plaintext) may be transformed into cryptograms (ciphertext) and back again.

One of the most basic kinds of cryptographic algorithms, employed since the time of the Romans, is called a substitution cipher. With this cipher, every letter of the plaintext alphabet is mapped to a different letter. For example, we might shift every letter to its successor: `A` becomes `B`, `B` becomes `C`, `Z` wraps around to become `A`, and so on. Thus, the plaintext

```
CLOS is an object-oriented programming language
```

may be enciphered to the cryptogram

```
DMPT jt bo pckfdu-psjfoufe qsphsbnnjoh mbohvbhf
```

Most often, the substitution of letters is jumbled. For example, `A` becomes `G`, `B` becomes `J`, and so on. As an example, consider the following cryptogram:

```
PDG TBCER CQ TCK AL S NGELCH QZBBR SBAJG
```

Hint: The letter `C` represents the plaintext letter `O`.

It is a vastly simplifying assumption to know that only a substitution cipher was employed to encode a plaintext message; nevertheless, deciphering the resulting cryptogram is not an algorithmically trivial task. Deciphering sometimes requires trial and error, wherein we make assumptions about a particular substitution and then evaluate their implications. For example, we may start with the one- and two-letter words in the cryptogram and hypothesize that they stand for common words such as `I` and `a`, or `it`, `in`, `is`, `of`, `or`, and `on`. By substituting the other occurrences of these ciphered letters, we may find hints for deciphering other words. For instance, if there is a three-letter word that starts with `o`, the word might reasonably be `one`, `our`, or `off`.

We can also use our knowledge of spelling and grammar to attack a substitution cipher. For example, an occurrence of double letters is not likely to represent the sequence `qq`. Similarly, we might try to expand a word ending with the letter `g` to the suffix `ing`. At a higher level of abstraction, we might assume that the sequence of words `it is` is more likely to occur than the sequence `if is`. Also, we might assume that the structure of a sentence typically includes a noun and a verb. Thus, if our analysis has identified a verb but no actor or agent, we might start a search for adjectives and nouns.

Sometimes we may have to backtrack. For example, we might have assumed that a certain two-letter word was `or`, but if the substitution for the letter `r` causes contradictions or blind alleys in other words, we might have to try the word `of` or `on` instead and consequently undo other assumptions we had based on this earlier substitution.

This leads us to the overarching requirement of our problem: to devise a system that, given a cryptogram, transforms it back to its original plaintext, assuming that only a simple substitution cipher was employed.

Defining the Boundaries of the Problem

As part of our analysis, let's walk through a scenario of solving a simple cryptogram. Spend the next few minutes solving the following problem, and as you proceed, record how you did it (no fair reading ahead!):

```
Q AZWS DSSC KAS DXZNN DASNN
```

As a hint, we note that the letter W represents the plaintext V.

Trying an exhaustive search is pretty much senseless. Assuming that the plaintext alphabet encompasses only the 26 uppercase English characters, there are 26! (approximately 4.03×10^{26}) possible combinations. Thus, we must try something other than a brute force attack. An alternate technique is to make an assumption based on our knowledge of sentence, word, and letter structure and then follow this assumption to its natural conclusions. Once we can go no further, we choose the next most promising assumption that builds on the first one, and so on, as long as each succeeding assumption brings us closer to a solution. If we find that we are stuck, or we reach a conclusion that contradicts a previous one, we must backtrack and alter an earlier assumption.

Here is our solution, showing the results at each step.

1. According to the hint, we may directly substitute V for W.
   ```
   Q AZVS DSSC KAS DXZNN DASNN
   ```
2. The first word is small, so it is probably either an A or an I; let's assume that it is an A.
   ```
   A AZVS DSSC KAS DXZNN DASNN
   ```
3. The third word needs a vowel, and it is likely to be the double letters. It is probably neither II nor UU, and it can't be AA because we have already used an A. Thus, we might try EE.
   ```
   A AZVE DEEC KAE DXZNN DAENN
   ```
4. The fourth word is three letters long and ends in an E; it is likely to be the word THE.
   ```
   A HZVE DEEC THE DXZNN DHENN
   ```
5. The second word needs a vowel, but only an I, O, or U (we've already used A and E). Only the I gives us a meaningful word.
   ```
   A HIVE DEEC THE DXINN DHENN
   ```
6. There are few four-letter words that have a double E, including DEER, BEER, and SEEN. Our knowledge of grammar suggests that the third word should be a verb, and so we select SEEN.
   ```
   A HIVE SEEN THE SXINN SHENN
   ```

7. This sentence is not making any sense (hives cannot see), so we probably made a bad assumption somewhere along the way. The problem seems to lie with the vowel in the second word, so we might consider reversing our initial assumption.

   I HAVE SEEN THE SXANN SHENN

8. Let's attack the last word. The double letters can't be SS (we've used an S, and besides, SHESS doesn't make any sense), but LL forms a meaningful word.

   I HAVE SEEN THE SXALL SHELL

9. The fifth word is part of a noun phrase and so is probably an adjective (STALL, for example, is rejected on this account). Searching for words that fit the pattern S?ALL yields SMALL.

   I HAVE SEEN THE SMALL SHELL

Thus, we have reached a solution.

We may make the following observations about this problem-solving process.

- We applied many different sources of knowledge, such as knowledge about grammar, spelling, and vowels.
- We recorded our assumptions in one central place and applied our sources of knowledge to these assumptions to reason about their consequences.
- We reasoned opportunistically. At times, we reasoned from general to specific rules (if the word is three letters long and ends in E, it is probably THE), and at other times, we reasoned from the specific to the general (?EE? might be DEER, BEER, or SEEN, but since the word must be a verb and not a noun, only SEEN satisfies our hypothesis).

From these problem-solving observations, we can identify some key abstractions. Key abstractions are analysis elements of our solution that begin to establish the initial architectural framework. The three bullets identify multiple knowledge sources, a central place for assumptions or hypotheses, and a control component that opportunistically controls the problem solving.

What we have described is a problem-solving approach known as a *blackboard model*. The blackboard model was first proposed by Newell in 1962 and later incorporated by Reddy and Erman into the Hearsay and Hearsay II projects, both of which dealt with the problems of speech recognition [4]. The blackboard model proved to be useful in this domain, and the framework was soon applied successfully to other domains, including signal interpretation, the modeling of three-dimensional molecular structures, image understanding, and planning [5]. Blackboard frameworks have proven to be particularly noteworthy with regard to the representation of declarative knowledge and are space and time efficient when compared with alternate approaches [6].

The blackboard framework is an architectural pattern that can be applied as a result of the analysis of our problem-solving algorithm. The framework can be represented in terms of classes and mechanisms that describe how instances of those classes collaborate.

The Architecture of the Blackboard Framework

Englemore and Morgan explain the blackboard model by analogy to the problem of a group of people solving a jigsaw puzzle:

> Imagine a room with a large blackboard and around it a group of people each holding over-size jigsaw pieces. We start with volunteers who put on the blackboard (assume it's sticky) their most "promising" pieces. Each member of the group looks at his pieces and sees if any of them fit into the pieces already on the blackboard. Those with the appropriate pieces go up to the blackboard and update the evolving solution. The new updates cause other pieces to fall into place, and other people go to the blackboard to add their pieces. It does not matter whether one person holds more pieces than another. The whole puzzle can be solved in complete silence; that is, there need be no direct communication among the group. Each person is self-activating, knowing when his pieces will contribute to the solution. No a priori established order exists for people to go up to the blackboard. The apparent cooperative behavior is mediated by the state of the solution on the blackboard. If one watches the task being performed, the solution is built incrementally (one piece at a time) and opportunistically (as an opportunity for adding a piece arises), as opposed to starting, say, systematically from the left top corner and trying each piece. [7]

As Figure 10–1 indicates, the blackboard framework consists of three elements: a blackboard, multiple knowledge sources, and a controller that mediates among these knowledge sources [8]. Notice how the following description describes the key abstractions identified from the problem space. According to Nii, "the purpose of the blackboard is to hold computational and solution-state data needed by and produced by the knowledge sources. The blackboard consists of objects from the solution space. The objects on the blackboard are hierarchically organized into levels of analysis. The objects and their properties define the vocabulary of the solution space" [9].

As Englemore and Morgan explain, "The domain knowledge needed to solve a problem is partitioned into knowledge sources that are kept separate and independent. The objective of each knowledge source is to contribute information that will lead to a solution to the problem. A knowledge source takes a set of current information on the blackboard and updates it as encoded in its specialized knowledge. The knowledge sources are represented as procedures, sets of rules, or logic assertions" [10].

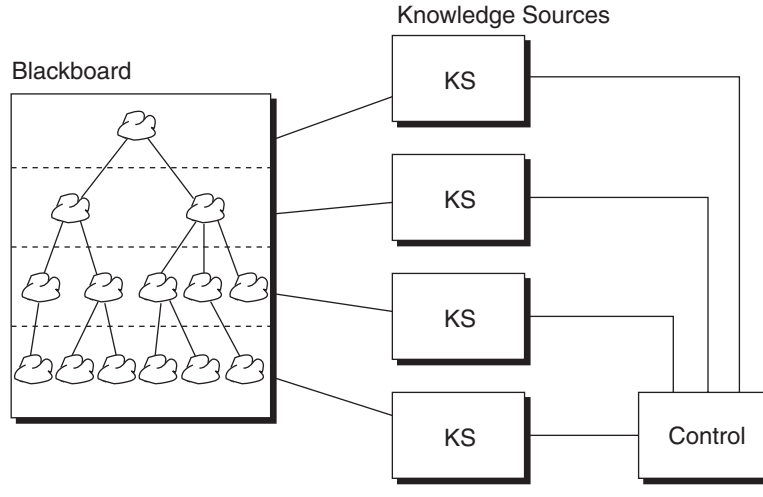

Figure 10–1 A Blackboard Framework

Knowledge sources, or KSs for short, are domain-specific. In speech recognition systems, knowledge sources might include agents that can reason about phonemes, morphemes, words, and sentences. In image recognition systems, knowledge sources would include agents that know about simple picture elements, such as edges and regions of similar texture, as well as higher-level abstractions representing the objects of interest in each scene, such as houses, roads, fields, cars, and people. Generally speaking, knowledge sources parallel the hierarchical structure of objects on the blackboard. Furthermore, each knowledge source uses objects at one level as its input and then generates and/or modifies objects at another level as its output. For instance, in a speech recognition system, a knowledge source that embodies knowledge about words might look at a stream of phonemes (at a low level of abstraction) to form a new word (at a higher level of abstraction). Alternately, a knowledge source that embodies knowledge about sentence structure might hypothesize the need for a verb (at a high level of abstraction); by filtering a list of possible words (at a lower level of abstraction), this knowledge source can verify the hypothesis.

These two approaches to reasoning represent forward-chaining and backward-chaining, respectively. Forward-chaining involves reasoning from specific assertions to a general assertion, and backward-chaining starts with a hypothesis, then tries to verify the hypothesis from existing assertions. This is why we say that control in the blackboard model is opportunistic: Depending on the circumstances, a knowledge source might be selected for activation that uses either forward- or backward-chaining.

Knowledge sources usually embody two elements, namely, preconditions and actions. The preconditions of a knowledge source represent the state of the blackboard in which the knowledge source shows an interest. For example, a precondition for a

knowledge source in an image recognition system might be the discovery of a relatively linear region of picture elements (perhaps representing a road). Triggering a precondition causes the knowledge source to focus its attention on this part of the blackboard and then take action by processing its rules or procedural knowledge.

Under these circumstances, polling is unnecessary: When a knowledge source thinks it has something interesting to contribute, it notifies the blackboard controller. Figuratively speaking, it is as if each knowledge source raises its hand to indicate that it has something useful to do; then, from among eager knowledge sources, the controller calls on the one that looks the most promising.

Analysis of Knowledge Sources

Let's return to our specific problem and consider the knowledge sources that can contribute to a solution. As is typical with most knowledge-engineering applications, the best strategy is to sit down with an expert in the domain and record the heuristics that this person applies to solve the problems in the domain. For our present problem, this might involve trying to solve a number of cryptograms and recording our thinking process along the way.

Our analysis suggests that 13 knowledge sources are relevant; they appear with the knowledge they embody in the following list:

- Common prefixes — Common word beginnings such as `re`, `anti`, and `un`
- Common suffixes — Common word endings such as `ly`, `ing`, `es`, and `ed`
- Consonants — Nonvowel letters
- Direct substitution — Hints given as part of the problem statement
- Double letters — Common double letters, such as `tt`, `ll`, and `ss`
- Letter frequency — Probability of the appearance of each letter
- Legal strings — Legal and illegal combinations of letters, such as `qu` and `zg`, respectively
- Pattern matching — Words that match a specified pattern of letters
- Sentence structure — Grammar, including the meanings of noun and verb phrases
- Small words — Possible matches for one-, two-, three-, and four-letter words
- Solved — Whether or not the problem is solved, or if no further progress can be made
- Vowels — Nonconsonant letters
- Word structure — The location of vowels and the common structure of nouns, verbs, adjectives, adverbs, articles, conjunctives, and so on

From an object-oriented perspective, each of these 13 knowledge sources represents a candidate class in our architecture: Each instance embodies some state (its knowledge), each exhibits certain class-specific behavior (a suffix knowledge source can react to words suspected of having a common ending), and each is uniquely identifiable (a small-word knowledge source exists independent of the pattern-matching knowledge source).

We may also arrange these knowledge sources in a hierarchy. Specifically, some knowledge sources operate on sentences, others on letters, still others on contiguous groups of letters, and the lowest-level ones on individual letters. Indeed, this hierarchy reflects the objects that may appear on the blackboard: sentences, words, strings of letters, and letters.

10.2 Elaboration

We are now ready to design a solution to the cryptanalysis problem using the blackboard framework we have described. This is a classic example of reuse-in-the-large, in that we are able to reuse a proven architectural pattern as the foundation of our design.

The architecture of the blackboard framework suggests that among the highest-level objects in our system are a blackboard, several knowledge sources, and a controller. Our next task is to identify the domain-specific classes and objects that specialize these general key abstractions.

Blackboard Objects

The blackboard is an elaborate structure of multiple levels of abstractions. The abstractions are captured as objects that appear hierarchically on a blackboard structure. The hierarchical object structure parallels the different levels of abstractions of the knowledge sources. The knowledge sources use the blackboard as a global source of input data, partial solutions, alternatives, final solutions, and control information.

To begin the design of the blackboard's hierarchical structure, we identify the following classes:

- `Sentence` A complete cryptogram
- `Word` A single word in the cryptogram
- `CipherLetter` A single letter of a word

Knowledge sources must also share knowledge about the assumptions each makes, so we include the following class of blackboard objects:

- `Assumption` An assumption made by a knowledge source

Finally, it is important to know what plaintext and ciphertext letters in the alphabet have been used in assumptions made by the knowledge sources, so we include the following class:

- `Alphabet` The plaintext alphabet, the ciphertext alphabet, and the mapping between the two

Is there anything in common among these five classes? We answer with a resounding yes: Each one of these classes represents objects that may be placed on a blackboard, and that very property distinguishes them from, for example, knowledge sources and controllers. Thus, we invent the `BlackboardObject` class as the superclass of every object that may appear on a blackboard. Figure 10–2 shows our preliminary design of the `Blackboard` abstraction.

Looking at the `BlackboardObject` class from its outside view, we may define two applicable operations:

- `register` Add the object to the blackboard.
- `resign` Remove the object from the blackboard.

Why do we define `register` and `resign` as operations on instances of `BlackboardObject`, instead of on the `Blackboard` itself? This situation is not unlike telling an object to draw itself in a window. The litmus test for deciding where to place these kinds of operations is whether or not the class itself has sufficient knowledge or responsibility to carry out the operation. In the case of

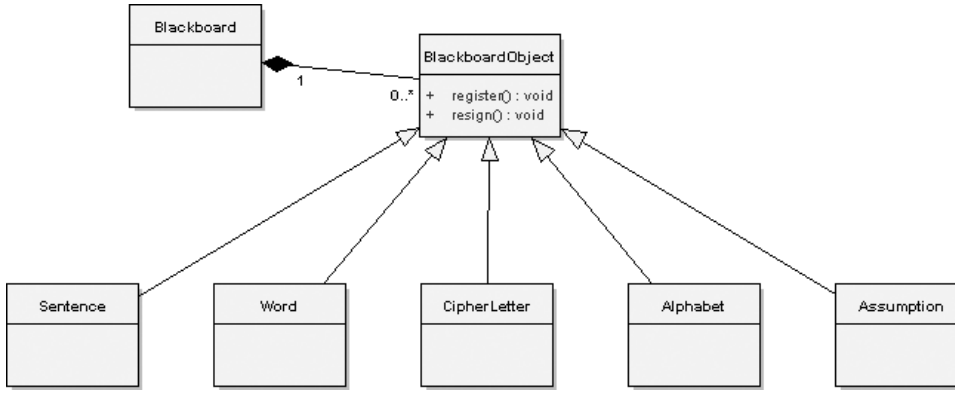

Figure 10–2 The Preliminary `Blackboard` Class Diagram Design

register and resign, this is indeed the case: The BlackboardObject is the only abstraction with detailed knowledge of how to attach or remove itself from the Blackboard (although it certainly does require collaboration with the BlackboardObject). In fact, it is an important responsibility of this abstraction that each BlackboardObject be self-aware as it is attached to the Blackboard because only then can it begin to participate in opportunistically solving the problem on the Blackboard.

Dependencies and Affirmations

Individual sentences, words, and cipher letters have another thing in common: Each has certain knowledge sources that depend on it. A given knowledge source may express an interest in one or more of these objects, and therefore, a sentence, word, or cipher letter must maintain a reference to each such knowledge source, so that when an assumption about the object changes, the appropriate knowledge sources can be notified that something interesting has happened. To provide this mechanism, we introduce a simple abstract class: Dependent.

To design the Dependent class, we include an object that represents a collection of knowledge sources:

- references Collection of knowledge sources

In addition, the following operations are defined for this class:

- add Add a reference to the knowledge source.
- remove Remove a reference to the knowledge source.
- numberOfDependents Return the number of dependents.
- notify Broadcast an operation of each dependent.

The operation notify has the semantics of a passive iterator, meaning that when we invoke it, we can supply an operation that we wish to perform on every dependent in the collection.

Dependency is an independent property that can be mixed in with other classes. For example, a CipherLetter is a BlackboardObject as well as a Dependent, so we can combine these two abstractions to achieve the desired behavior. Using an abstract class in this way increases the reusability and separation of concerns in our architecture.

CipherLetter and Alphabet have another property in common: Instances of both of these classes may have assumptions made about them (and remember that an Assumption object is also a kind of BlackboardObject). For

example, a certain knowledge source might assume that the ciphertext letter K represents the plaintext letter P. As we get closer to solving our problem, we might make the unchangeable assertion that G represents J. Thus we need to include a class that maintains the assumptions and assertions about the associated object. This class we will identify as Affirmation.

In our architecture, we will make affirmations only about individual letters, as in CipherLetter and Alphabet. As our earlier scenario implied, cipher letters represent single letters about which statements might be made, and alphabets comprise many letters, each of which might have different statements made about them. Defining Affirmation as an independent class thus captures the common behavior across these two disparate classes.

We define the following operations for instances of the Affirmation class:

- make Make a statement.
- retract Retract a statement.
- ciphertext Given a plaintext letter, return its ciphertext equivalent.
- plaintext Given a ciphertext letter, return its plaintext equivalent.

Further analysis suggests that we should clearly distinguish between the two roles played by a statement: An assumption, which represents a temporary mapping between a ciphertext letter and its plaintext equivalent, and an assertion, which is a permanent mapping, meaning that the mapping is defined and therefore not changeable. During the solution of a cryptogram, knowledge sources will make many assumptions, and as we move closer to a final solution, these mappings eventually become assertions. To model these changing roles, we will refine the previously identified class Assumption and introduce a new subclass named Assertion, both of whose instances are managed by instances of the class Affirmation as well as placed on the blackboard. We begin by completing the signature of the operations make and retract to include an Assumption or Assertion argument, and then add the following selectors:

- isPlainLetterAsserted A selector: Is the plaintext letter defined?
- isCipherLetterAsserted A selector: Is the ciphertext letter defined?
- plainLetterHasAssumption A selector: Is there an assumption about the plaintext letter?
- cipherLetterHasAssumption A selector: Is there an assumption about the ciphertext letter?

`Assumption` objects are a kind of `BlackboardObject` because they represent state that is of general interest to all knowledge sources. Member objects will need to be declared to represent the following properties:

- `target` The blackboard object about which the assumption was made
- `creator` The knowledge source that created the assumption
- `reason` The reason the knowledge source made the assumption
- `plainLetter` The plaintext letter about which the assumption is being made
- `cipherLetter` The assumed value of the plaintext letter

The need for each of these properties is largely derived from the very nature of an assumption: A particular knowledge source makes an assumption about a plaintext/ciphertext mapping and does so for a certain reason (usually because some rule was triggered). The need for the first member, `target`, is less obvious. We include it because of the problem of backtracking. If we ever have to reverse an assumption, we must notify all blackboard objects for which the assumption was originally made, so that they in turn can alert the knowledge sources they depend on (via the dependency mechanism) that their meaning has changed.

Next, we declare the subclass of `Assumption` named `Assertion`. The classes `Assumption` and `Assertion` share the following operation, among others:

- `isRetractable` A selector: Is the mapping temporary?

All `Assumption` objects answer true to the predicate `isRetractable`, whereas all `Assertion` objects answer false. Additionally, once made, an assertion can neither be restated nor retracted.

Figure 10–3 provides a class diagram that illustrates the collaboration of the `Dependent` and `Affirmation` classes. Pay particular attention to the roles each abstraction plays in the various associations. For example, a `KnowledgeSource` is the `creator` of an `Assumption` and is also the `referencer` of a `CipherLetter`. Because a role represents a different view than an abstraction presents to the world, we would expect to see a different protocol between knowledge sources and assumptions than between knowledge sources and letters.

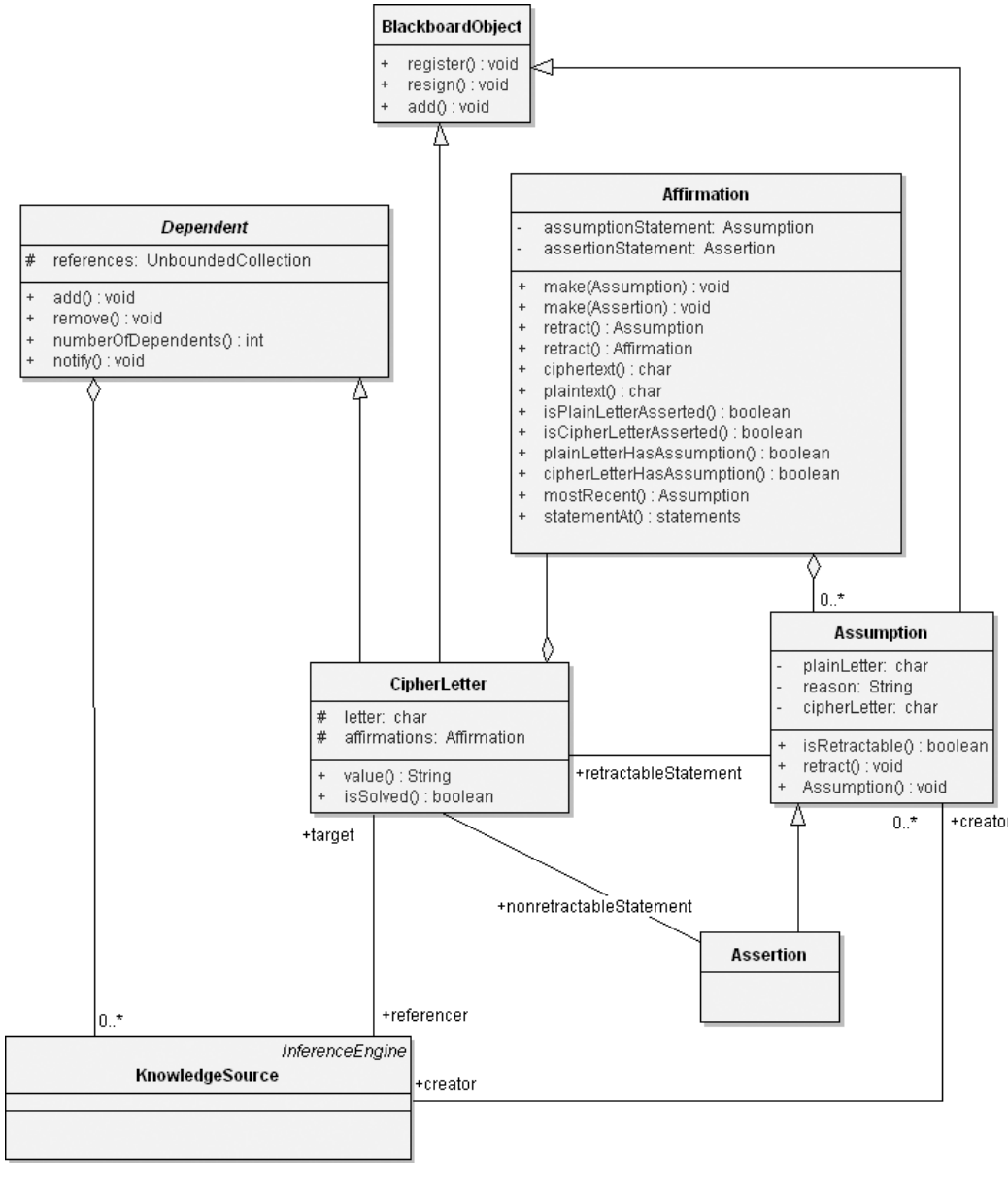

Figure 10–3 Dependency and Affirmation Classes

10.3 Construction

Let's continue our design of the `Sentence`, `Word`, and `CipherLetter` classes, followed by the `Alphabet` class, by doing a little isolated class design.

Designing the Blackboard Objects

A sentence is quite simple: It is a `BlackboardObject` as well as a `Dependent`, and it denotes a list of words that compose the sentence.

We make the superclass `Dependent` abstract[1] (Figure 10–4) because we expect there may be other `Sentence` subclasses that try to inherit from `Dependent` as well. By marking this inheritance relationship abstract, we cause such subclasses to share a single `Dependent` superclass.

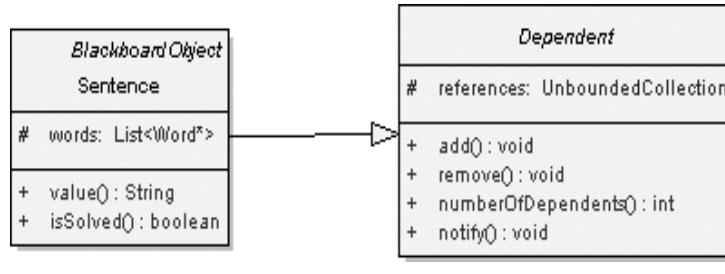

Figure 10–4 The `Sentence` Class Design with the Abstract `Dependent` Class

In addition to the operations `register` and `resign` defined by its superclass `BlackboardObject`, plus the four operations defined in `Dependent`, we add the following two sentence-specific operations:

- `value` Return the current value of the sentence.
- `isSolved` Return true if there is an assertion for all words in the sentence.

At the start of the problem, `value` returns a string representing the original cryptogram. Once `isSolved` evaluates as true, the operation `value` may be used to retrieve the plaintext solution. Accessing `value` before `isSolved` is true will yield partial solutions.

1. In UML 2.0, an abstract class is represented with the class name in italics. A keyword {`abstract`} may also be placed in the property list.

Just like the Sentence class, a Word is a kind of BlackboardObject as well as a kind of Dependent. Furthermore, a Word denotes a list of letters. To assist the knowledge sources that manipulate words, we include a reference from a word to its sentence, as well as from a word to the previous and next words in the sentence.

As we did for the Sentence operations, we define the following two operations for the class Word:

- value Return the current value of the word.
- isSolved Return true if there is an assertion for every letter in the word.

We may next define the class CipherLetter. An instance of this class is a kind of BlackboardObject and a kind of Dependent. In addition to its inherited behaviors, each CipherLetter object has a value (such as the ciphertext letter H) together with a collection of assumptions and assertions regarding its corresponding plaintext letter. We can use the class Affirmation to collect these statements. Figure 10–5 illustrates the addition of the design of CipherLetter and Word in our architecture framework.

Notice that we include the selectors value and isSolved, similar to our design of Sentence and Word. We must also eventually provide operations for the clients of CipherLetter to access its assumptions and assertions in a safe manner.

One comment about the member object affirmations: We expect this to be a collection of assumptions and assertions ordered according to their time of creation, with the most recent statement in this collection representing the current assumption or assertion. The reason we choose to keep a history of all assumptions is to permit knowledge sources to look at earlier assumptions that were rejected, so that they can learn from earlier mistakes. This decision influences our design decisions about the Affirmation class, to which we add the following operations:

- mostRecent A selector: returns the most recent assumption or assertion
- statementAt A selector: returns the nth statement

Now that we have refined its behavior, we can next make a reasonable implementation decision about the Affirmation class. Specifically, we can include the protected member object statements, which is defined as a collection of assumptions.

Consider next the class named Alphabet. This class represents the entire plaintext and ciphertext alphabet, plus the mappings between the two. This information is important because each knowledge source can use it to determine which

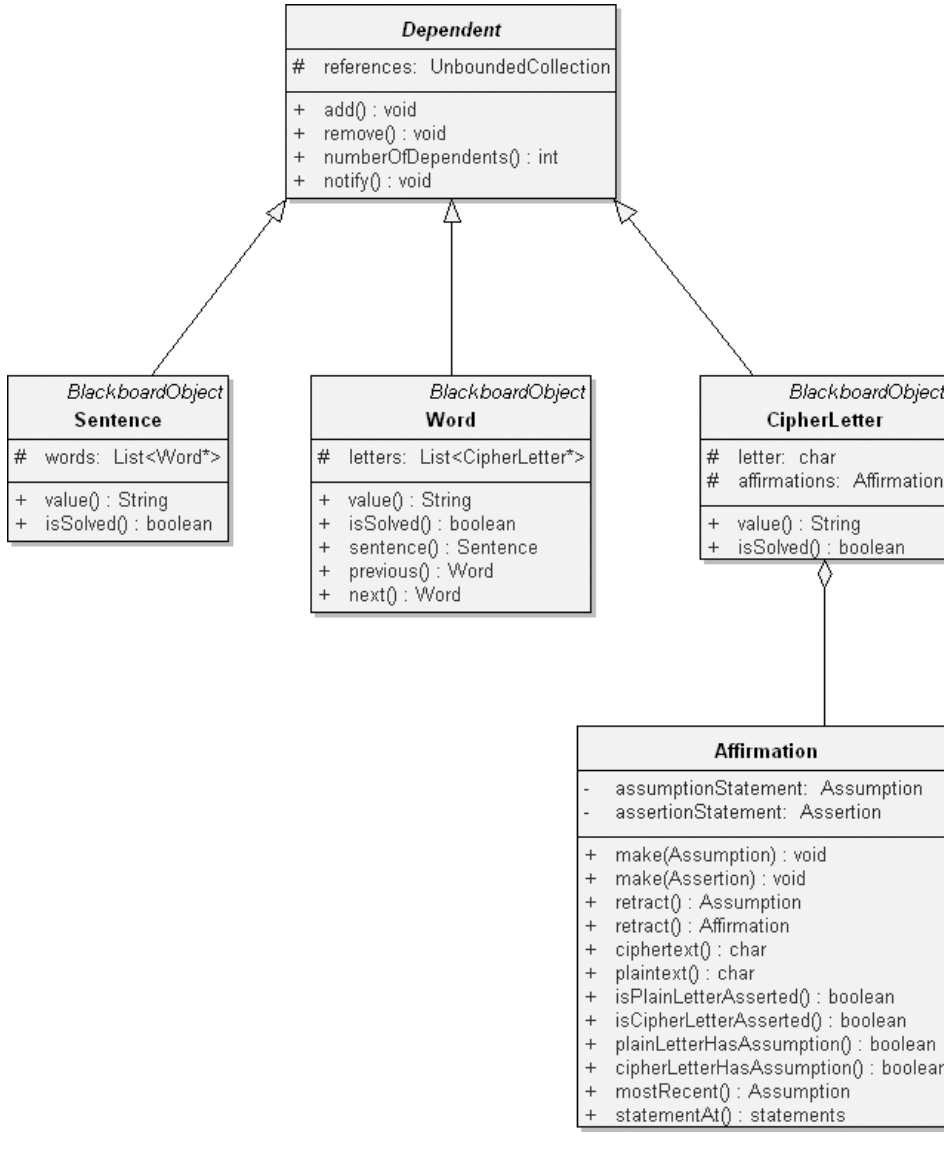

Figure 10–5 The Design of the `CipherLetter` and `Word` Classes

mappings have been made and which are yet to be done. For example, if we already have an assertion that the ciphertext letter `C` is really the letter `M`, then an alphabet object records this mapping so that no other knowledge source can apply the plaintext letter `M`. For efficiency, we need to query about the mapping both ways: Given a ciphertext letter, return its plaintext mapping, and given a plaintext letter, return its ciphertext mapping. Figure 10–6 illustrates the addition of the design of the `Alphabet` class.

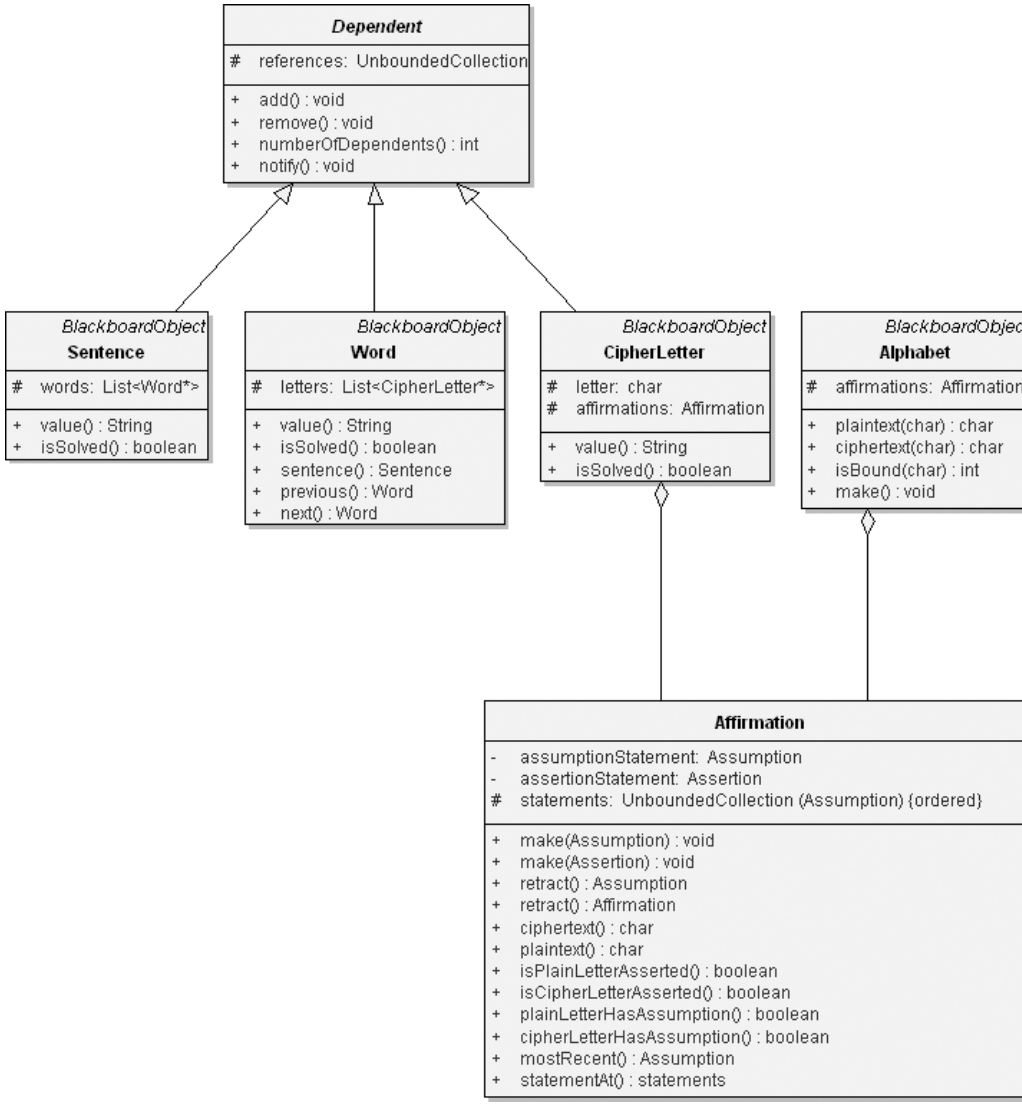

Figure 10–6 The Design of the `Alphabet` Class

Just as for the `CipherLetter` class, we also include a protected member object `affirmations` and provide suitable operations to access its state.

Now we are ready to define the `Blackboard` class. This class has the simple responsibility of collecting instances of the `BlackboardObject` class and its subclasses. Thus we may design `Blackboard` as a type of instance of a `DynamicCollection`. We have chosen to inherit from rather than contain an

instance of the DynamicCollection class because Blackboard passes our test for inheritance: A Blackboard is indeed a kind of collection.

The Blackboard class provides operations such as add and remove, which it inherits from the Collection class. Our design includes five operations specific to the blackboard.

- reset — Clean the blackboard.
- assertProblem — Place an initial problem on the blackboard.
- connect — Attach the knowledge source to the blackboard.
- isSolved — Return true if the sentence is solved.
- retrieveSolution — Return the solved plaintext sentence.

The second operation is needed to create a dependency between a blackboard and its knowledge sources.

In Figure 10–7, we summarize our design of the classes that collaborate with Blackboard. In this diagram, notice that we show the Blackboard class as both instantiating and inheriting from the template class DynamicCollection. This diagram also clearly shows why introducing the Dependent class as an abstract class was a good design decision. Specifically, Dependent represents a behavior that encompasses only a partial set of BlackboardObject subclasses. By making Dependent abstract, we increase its chances of being reused.

Designing the Knowledge Sources

In a previous section, we identified 13 knowledge sources relevant to this problem. Just as we did for the Blackboard objects, we can design a class structure encompassing these knowledge sources and thereby elevate all common characteristics to more abstract classes.

Designing Specialized Knowledge Sources

Assume for the moment the existence of an abstract class called KnowledgeSource, whose purpose is much like that of the class BlackboardObject. Rather than treat each of the 13 knowledge sources as a direct subclass of this more general class, it is useful to first perform a domain analysis and see if there are any clusters of knowledge sources. Indeed, there are such groups: Some knowledge sources operate on whole sentences, others on whole words, others on contiguous strings of letters, and still others on individual letters. We may capture these design decisions by creating the following subclasses:

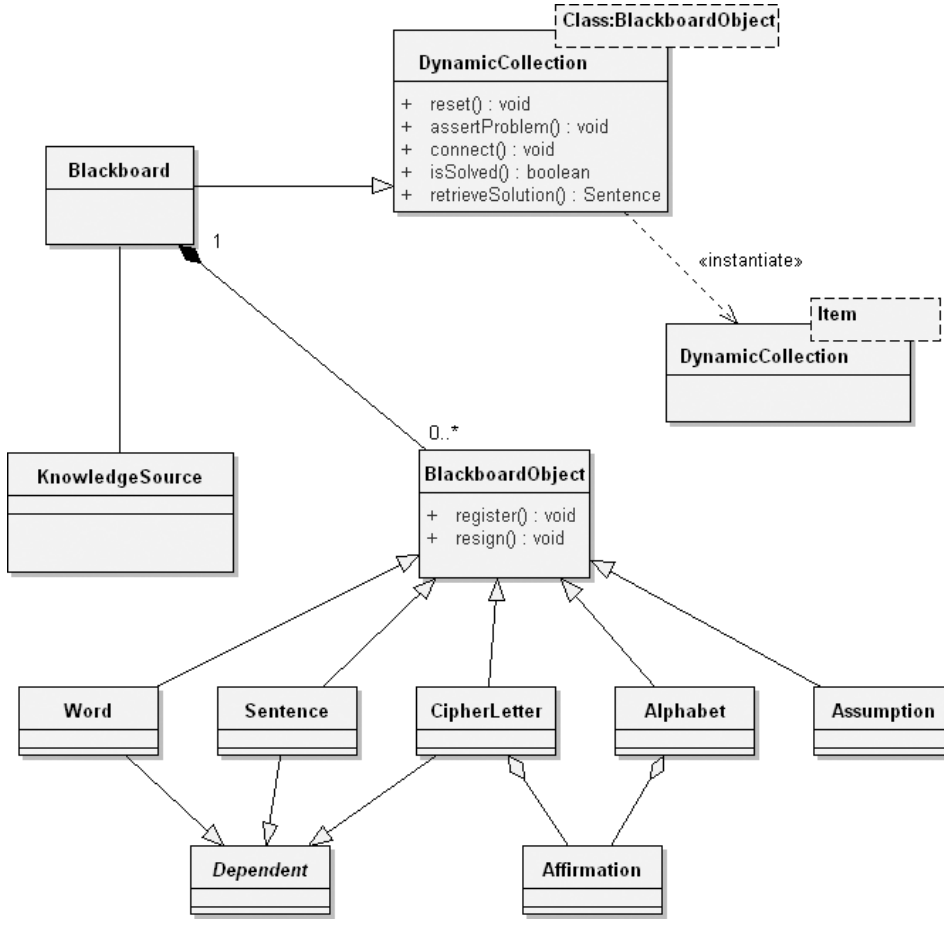

Figure 10–7 The Refined `Blackboard` Class Diagram Design

- `SentenceKnowledgeSource` Rules associated with sentences
- `WordKnowledgeSource` Rules associated with words
- `LetterKnowledgeSource` Rules associated with letters
- `StringKnowledgeSource` Rules associated with strings

For each of these classes, we may provide another level of specification. For example, the subclasses of the class `SentenceKnowledgeSource` include the following:

- `SentenceStructureKnowledgeSource` Rules specific to sentence structure
- `SolvedKnowledgeSource` Solved cryptogram sentence

Similarly, the subclasses of the intermediate class `WordKnowledgeSource` include these:

- `WordStructureKnowledgeSource` — Rules specific to word structure
- `SmallWordKnowledgeSource` — Rules specific to small words
- `PatternMatchingKnowledgeSource` — Rules for matching patterns of words

The last class requires some explanation. In our earlier list of the 13 knowledge sources, we said that the purpose of a pattern-matching knowledge source was to propose words that fit a certain pattern. We can use regular expression pattern-matching symbols such as:

- Any item `?`
- Not item `~`
- Closure item `*`
- Start group `{`
- Stop group `}`

With these symbols, we might give an instance of this class the pattern `?E~{A E I O U}`, thereby asking it to give us from its dictionary all the three-letter words starting with any letter, followed by an `E`, and ending with any letter except a vowel. All instances of this class share a dictionary of words, and each instance has its own regular expression pattern-matching agent. The detailed behavior of this class is not important to us at this point in our design, so we will defer the invention of the remainder of its interface and implementation.

Continuing, we may declare the following subclasses of the class `String KnowledgeSource`:

- `CommonPrefixKnowledgeSource` — Rules specific to prefixes
- `CommonSuffixKnowledgeSource` — Rules specific to suffixes
- `DoubleLetterKnowledgeSource` — Rules for double letters, e.g., `oo`, `ll`, and so on
- `LegalStringKnowledgeSource` — Rules specific to what makes legal strings

Lastly, we can introduce the following subclasses of the class `LetterKnowledgeSource`:

- `DirectSubstitutionKnowledgeSource` Rules specific to substitution of letters
- `VowelKnowledgeSource` Rules specific for vowels
- `ConsonantKnowledgeSource` Rules specific for consonants
- `LetterFrequencyKnowledgeSource` Rules specific to frequency of letters

Figure 10–8 illustrates the hierarchical structure of `KnowledgeSource`.

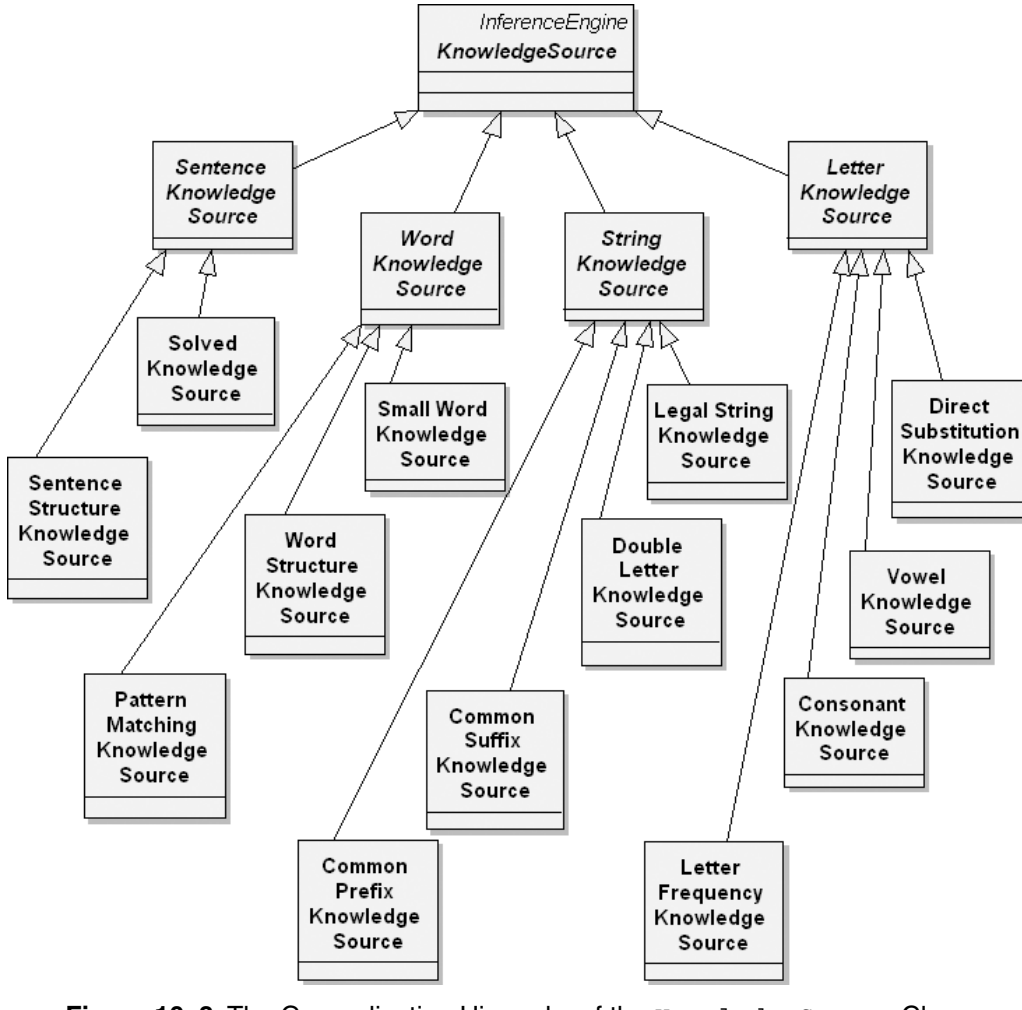

Figure 10–8 The Generalization Hierarchy of the `KnowledgeSource` Class

Generalizing the Knowledge Sources

Analysis suggests that only two primary operations apply to all these specialized classes:

- `reset` Restart the knowledge source.
- `evaluate` Evaluate the state of the blackboard.

The reason for this simple interface is that knowledge sources are relatively autonomous entities: We point one to an interesting `Blackboard` object and then tell it to evaluate its rules according to the current global state of the `Blackboard`. As part of the evaluation of its rules, a given knowledge source might do any one of several things.

- Propose an assumption about the substitution cipher.
- Discover a contradiction among previous assumptions, and cause the offending assumption to be retracted.
- Propose an assertion about the substitution cipher.
- Tell the controller that it has some interesting knowledge to contribute.

These are all general actions that are independent of the specific kind of knowledge source. To generalize even further, these actions represent the behavior of an inference engine. Therefore, we create the class `InferenceEngine` that, given a set of rules, evaluates those rules either to generate new rules (forward-chaining) or to prove some hypothesis (backward-chaining). When designing the constructor for `InferenceEngine`, the basic responsibility is to create an instance of this class and populate it with a set of rules, which it then uses for evaluation.

In fact, this class has only one critical operation that it makes visible to knowledge sources:

- `evaluate` Evaluate the rules of the inference engine.

This then is how knowledge sources collaborate: Each specialized knowledge source defines its own knowledge-specific rules and delegates responsibility for evaluating these rules to the `InferenceEngine` class. More precisely, we may say that the operation `KnowledgeSource::evaluate` ultimately invokes the operation `InferenceEngine::evaluate`, the results of which are used to carry out any of the four actions we listed earlier. In Figure 10–9, we illustrate a common scenario of this collaboration.

The sequence diagram illustrates the following steps in this scenario:

1. Select a `KnowledgeSource` for action.
2. Evaluate the `KnowledgeSource` against the state of the `Blackboard`.

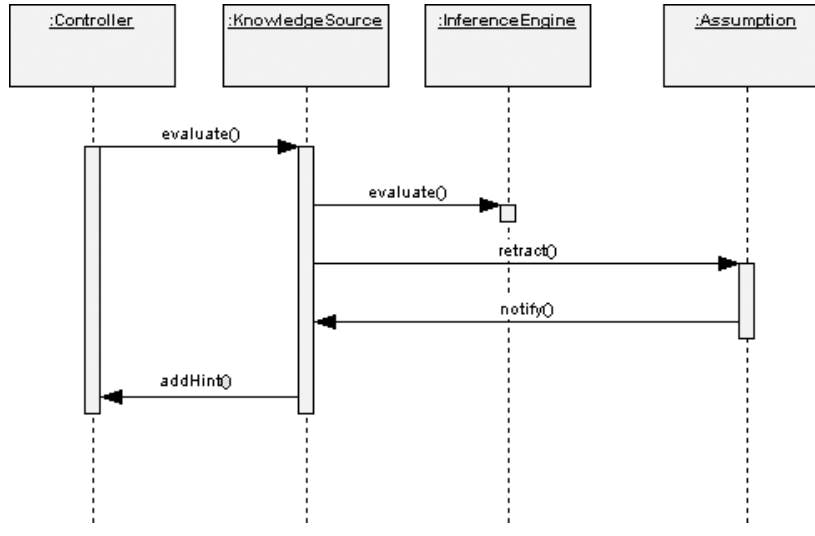

Figure 10–9 A Scenario for Evaluating Knowledge Source Rules

3. Take some action, such as retracting an `Assumption`.
4. Notify all dependent `KnowledgeSource` objects that the assumption has been retracted.
5. Tell the `Controller` that the `KnowledgeSource` has a new hint to offer in solving the blackboard problem.

What exactly is a rule? A rule might be composed for the common suffix knowledge source using a pattern-matching algorithm that recognizes a common suffix pattern such as `*I??`. Given a string of letters matching the regular expression pattern `*I??`, the candidate suffixes may include `ING`, `IES`, and `IED`.

In terms of its class structure, we may thus say that a knowledge source is a kind of inference engine. Additionally, each knowledge source must have some association with a blackboard object, for that is where it finds the objects on which it operates. Finally, each knowledge source must have an association to a controller, with which it collaborates by sending hints of solutions; in turn, the controller might trigger the knowledge source from time to time. Figure 10–10 illustrates these design decisions.

We also introduce the collection `pastAssumptions`, so that the knowledge source can keep track of all the assumptions and assertions it has ever made, in order to learn from its mistakes.

Instances of the `Blackboard` class serve as a repository of `Blackboard` objects. For a similar reason, we need a `KnowledgeSources` class, denoting the entire collection of knowledge sources for a particular problem.

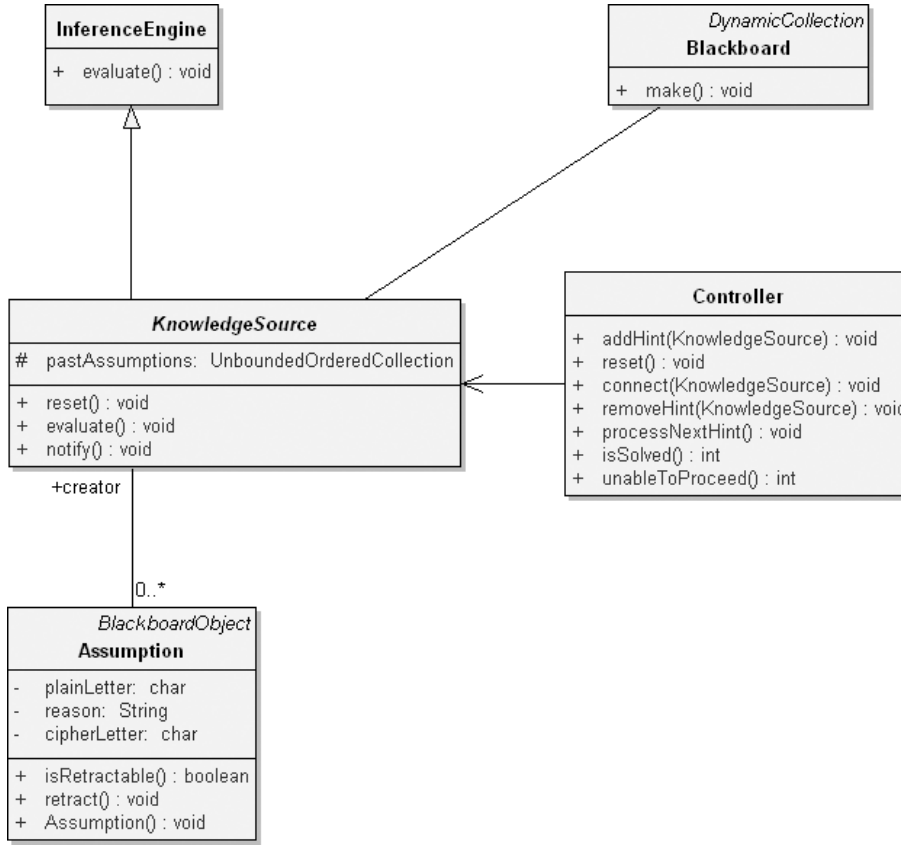

Figure 10–10 The Preliminary Design of `KnowledgeSource`

One of the responsibilities of this class is that when we create an instance of `KnowledgeSources`, we also create the 13 individual `KnowledgeSource` objects. We may perform three operations on instances of this class:

- `restart` — Restart the knowledge sources.
- `startKnowledgeSource` — Give a specific knowledge source its initial conditions.
- `connect` — Attach the knowledge source to the blackboard or to the controller.

Figure 10–11 provides the refined design of the class structure of the `KnowledgeSource` classes, according to these design decisions.

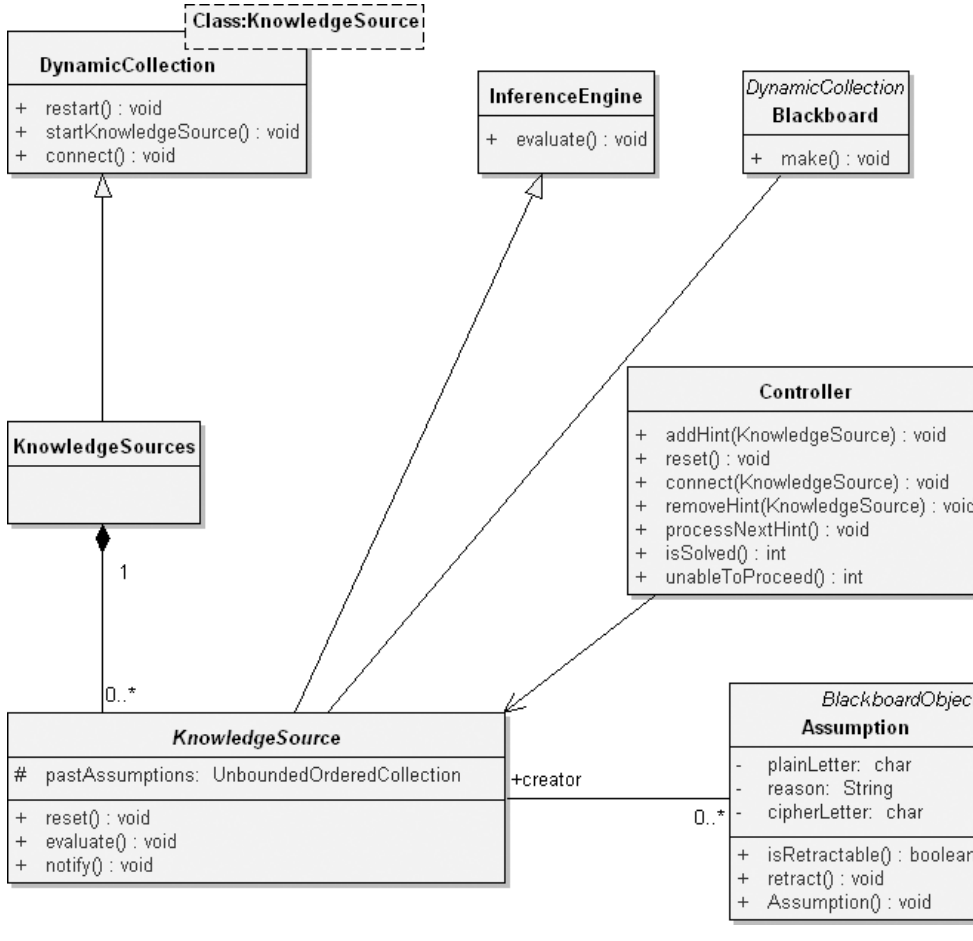

Figure 10–11 The Refined Design of the `KnowledgeSource` Class Diagram

Designing the Controller

Consider for a moment how the controller and individual knowledge sources interact. At each stage in the solution of a cryptogram, a particular knowledge source might discover that it has a useful contribution to make and so gives a hint to the controller. Conversely, the knowledge source might decide that its earlier hint no longer applies and so may remove the hint. Once all knowledge sources have been given a chance, the controller selects the most promising hint and activates the appropriate knowledge source by invoking its `evaluate` operation.

How does the controller decide which knowledge source to activate? We may devise a few suitable rules.

- An `Assertion` has a higher priority than an `Assumption`.
- The `SolvedKnowledgeSource` provides the most useful hints.
- The `PatternMatchingKnowledgeSource` provides higher-priority hints than the `SentenceStructureKnowledgeSource`.

A controller thus acts as an agent responsible for mediating among the various knowledge sources that operate on a blackboard.

The controller must have an association to its knowledge sources, which it can access through the appropriately named class `KnowledgeSources`. Additionally, the controller must have as one of its properties a collection of hints, ordered in accordance with the above rules of prioritization. In this manner, the controller can easily select for activation the knowledge source with the most interesting hint to offer.

Engaging in a little more isolated class design, we offer the following operations for the `Controller` class.

■ `reset`	Restart the controller.
■ `addHint`	Add a knowledge source hint.
■ `removeHint`	Remove a knowledge source hint.
■ `processNextHint`	Evaluate the next-highest-priority hint.
■ `isSolved`	A selector: Return true if the problem is solved.
■ `unableToProceed`	A selector: Return true if the knowledge sources are stuck.
■ `connect`	Attach the controller to the knowledge source.

The controller is in a sense driven by the hints it receives from various knowledge sources. As such, finite state machines are well suited for capturing the dynamic behavior of this class.

For example, consider the state transition diagram shown in Figure 10–12. Here we see that a controller may be in one of five major states: `Initializing`, `Selecting`, `Evaluating`, `Stuck`, and `Solved`. The controller's most interesting activity occurs between the `Selecting` and `Evaluating` states. While selecting, the controller naturally transitions from the state `Creating Strategy` to `Processing Hint` and eventually to `Selecting KS`. If a knowledge source is in fact selected, then the controller transitions to the `Evaluating` state, wherein it first is in `Updating Blackboard`. It transitions to `Connecting` if objects are added and to `Backtracking` if assumptions are retracted, at which time it also notifies all dependents.

The controller unconditionally transitions to `Stuck` if it cannot proceed and to `Solved` if it finds a solved blackboard problem.

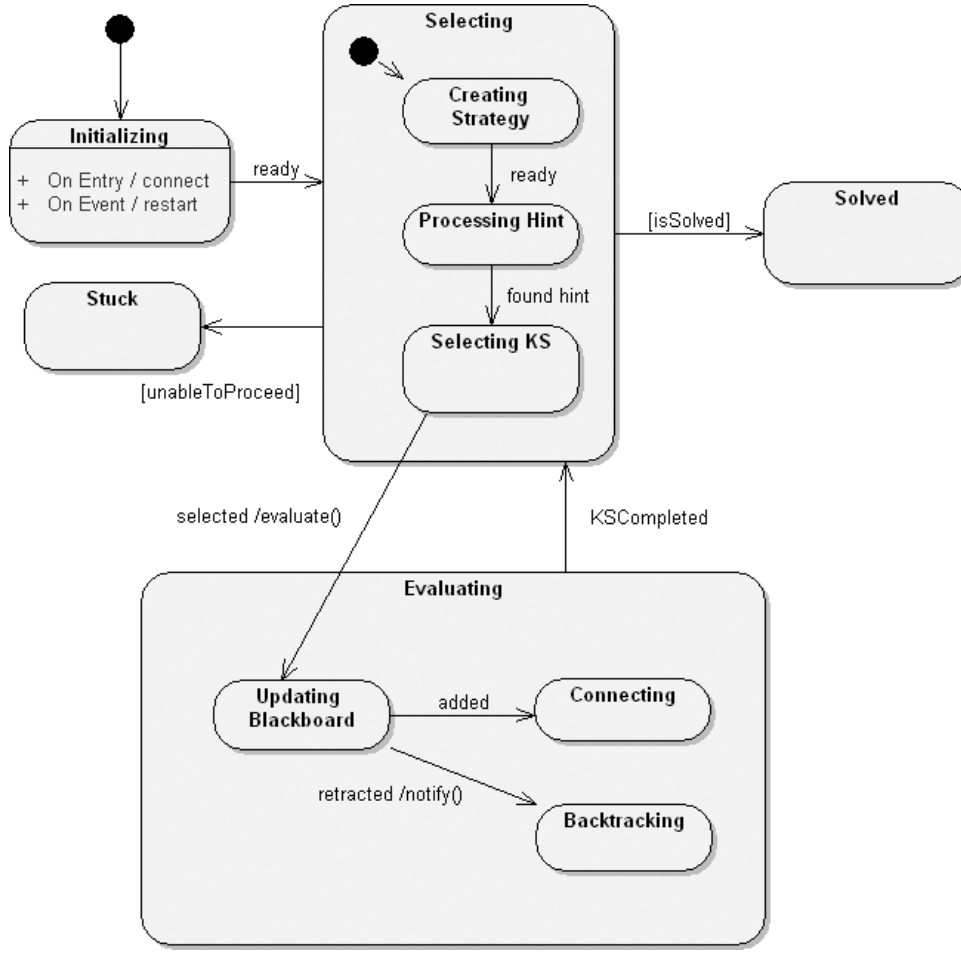

Figure 10–12 The Controller State Machine

Integrating the Blackboard Framework

Now that we have defined the key abstractions for our domain, we may continue by putting them together to form a complete application. We will proceed by implementing and testing a vertical slice through the architecture and then by completing the system one mechanism at a time.

Integrating the Topmost Objects

Figure 10–13 is a composite structure diagram that captures our design of the topmost object in the system, paralleling the structure of the generic blackboard

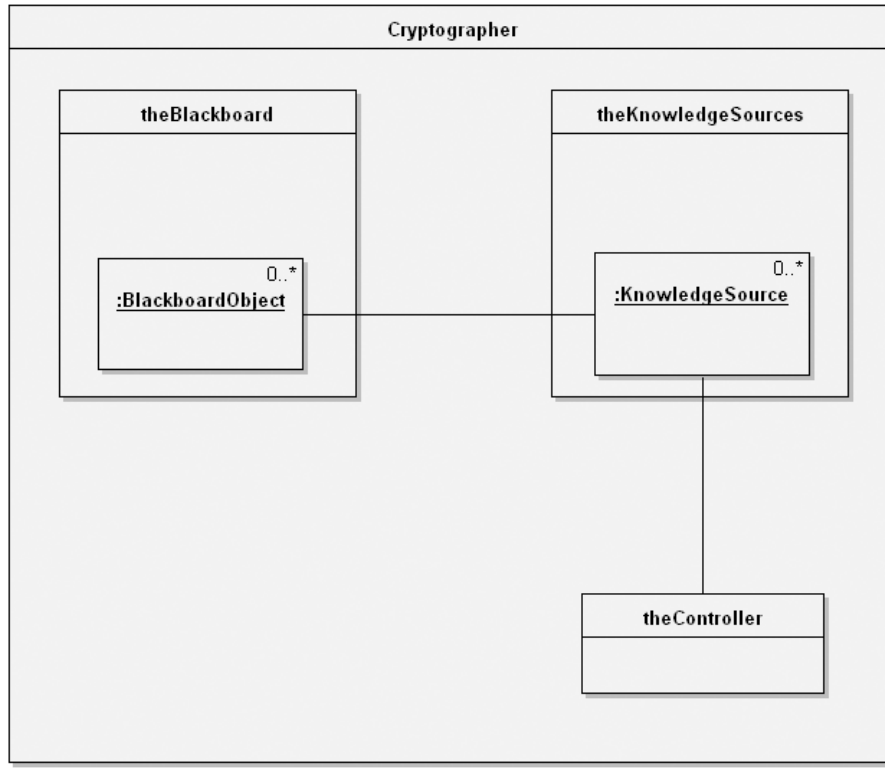

Figure 10–13 The Cryptanalysis Composite Structure Diagram

framework shown earlier in Figure 10–1. In Figure 10–13, we show the physical containment of blackboard objects by the collection `theBlackboard` and knowledge sources by the collection `theKnowledgeSources`, using a shorthand style identical to that for showing nested classes.

In this diagram, we introduce an instance of a new class that we call `Cryptographer`. The intent of this class is to serve as an aggregate encompassing the blackboard, the knowledge sources, and the controller. In this manner, our application might provide several instances of this class and thus have several blackboards running simultaneously.

We define two primary operations for the `Cryptographer` class:

- `reset` Restart the blackboard.
- `decipher` Solve the given cryptogram.

The behavior we require as part of this class's constructor is to create the dependencies between the blackboard and its knowledge sources, as well as between

the knowledge sources and the controller. The `reset` method is similar, in that it simply resets these connections and returns the blackboard, the knowledge sources, and the controller back to a stable initial state.

Although we will not show its details here, the signature of the `decipher` operation includes a string, through which we provide the ciphertext to be solved. In this manner, the root of our main program becomes embarrassingly simple, as is common in well-designed object-oriented systems.

The implementation of the `decipher` operation is, not surprisingly, slightly more complicated. Basically, we must first invoke the `assertProblem` operation to set up the problem on the blackboard. Next, we must start the knowledge sources by bringing their attention to this new problem. Finally, we must loop, telling the controller to process the next hint at each new pass, either until the problem is solved or until all the knowledge sources are unable to proceed. Figure 10–14 illustrates the flow of control using a sequence diagram.

We would be best advised to complete enough of the relevant architectural interfaces so that we could complete this algorithm and execute it. Although at this point it would have minimal functionality, its implementation as a vertical slice

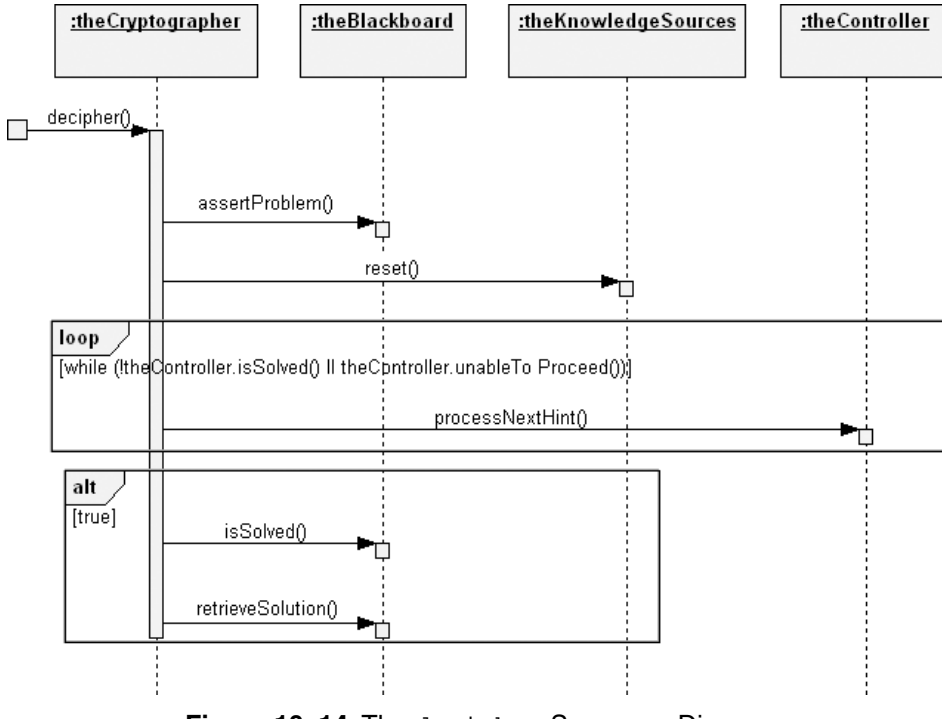

Figure 10–14 The `decipher` Sequence Diagram

through the architecture would force us to validate certain key architectural decisions.

Continuing, let's look at two of the key operations used in `decipher`, namely, `assertProblem` and `retrieveSolution`. The `assertProblem` operation is particularly interesting because it must generate an entire set of `Blackboard` objects. In the form of a simple pseudocode script, our algorithm is as follows.

```
trim all leading and trailing blanks from the string
return if the resulting string is empty
create a sentence object
add the sentence to the blackboard
create a word object (this will be the leftmost word in the
  sentence)
add the word to the blackboard
add the word to the sentence
for each character in the string, from left to right
  if the character is a space
    make the current word the previous word
      create a word object
      add the word to the blackboard
      add the word to the sentence
    else
      create a cipher-letter object
      add the letter to the blackboard
      add the letter to the word
```

The purpose of design is simply to provide a blueprint for implementation. This script supplies a sufficiently detailed algorithm, so we need not show its complete implementation.

The operation `retrieveSolution` is far simpler; we simply return the value of the sentence on the blackboard. Calling `retrieveSolution` before `isSolved` evaluates true will yield partial solutions.

Implementing the Assumption Mechanism

At this point, we have implemented the mechanisms that allow us to set and retrieve values for `Blackboard` objects. The next major function point involves the mechanism for making assumptions about `Blackboard` objects. This is a particularly significant issue because assumptions are dynamic (meaning that they are routinely created and destroyed during the process of forming a solution), and their creation or retraction triggers controller events.

Figure 10–15 illustrates the primary scenario of when a knowledge source states an assumption. As this communication diagram shows, once the `KnowledgeSource` creates an `Assumption`, it notifies the `Blackboard`, which in turn makes the `Assumption` for its `Alphabet` and then for each `BlackboardObject` to which the `Assumption` applies. Using the dependency mechanism, the affected `BlackboardObject` in turn might notify any dependent `KnowledgeSource`.

In its most naive implementation, retracting an assumption simply undoes the work of this mechanism. For example, to retract an assumption about a cipher letter, we just pop its collection of assumptions, up to and including the assumption we are retracting. In this manner, the given assumption and all assumptions that built on it are undone.

A more sophisticated mechanism is possible. For example, suppose that we made an assumption that a certain one-letter word is really just the letter `I` (assuming we need a vowel). We might make a later assumption that a certain double-letter word is `NN` (assuming we need a consonant). If we then find we must retract the first assumption, we probably don't have to retract the second one. This approach requires us to add a new behavior to the `Assumption` class so that it can keep track of what assumptions are dependent on others. We can reasonably defer this enhancement until much later in the evolution of this system because adding this behavior has no architectural impact.

Adding New Knowledge Sources

Now that we have the key abstractions of the blackboard framework in place, and once the mechanisms for stating and retracting assumptions are working, our next

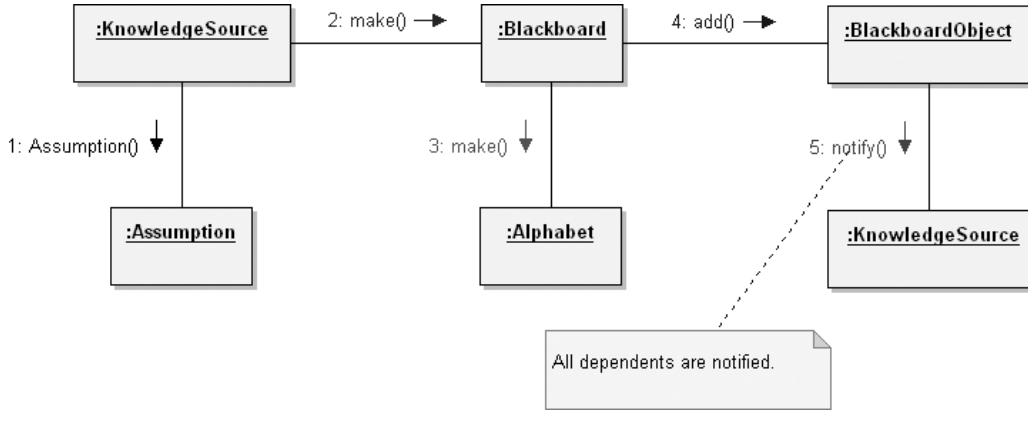

Figure 10–15 The Assumption Mechanism

step is to implement the `InferenceEngine` class since all knowledge sources depend on it. As we mentioned earlier, this class has only one really interesting operation, namely, `evaluate`. We will not show its details here because this particular method reveals no new important design issues.

Once we are confident that our inference engine works properly, we may incrementally add each knowledge source. We emphasize the use of an incremental process for two reasons.

1. For a given knowledge source, it is not clear what rules are really important until we apply them to real problems.
2. Debugging the knowledge base is far easier if we implement and test smaller related sets of rules, rather than trying to test them all at once.

Fundamentally, implementing each knowledge source is largely a problem of knowledge engineering. For a given knowledge source, we must confer with an expert (perhaps a cryptologist) to decide which rules are meaningful. As we test each knowledge source, our analysis may reveal that certain rules are useless, others are either too specific or too general, and perhaps some are missing. We may then choose to alter the rules of a given knowledge source or even add new sources of knowledge.

As we implement each knowledge source, we may discover the existence of common rules as well as common behavior. For example, we might notice that the `WordStructureKnowledgeSource` and the `SentenceStructure KnowledgeSource` classes share a common behavior, in that both must know how to evaluate rules regarding the legal ordering of certain constructs. The former knowledge source is interested in the arrangement of letters; the latter is interested in the arrangement of words. In either case, the processing is the same. Thus, it is reasonable for us to alter the knowledge source class structure by developing a new abstract class, called `StructureKnowledgeSource`, in which we place this common behavior.

This new knowledge source class hierarchy highlights the fact that evaluating a set of rules is dependent on both the kind of knowledge source as well as the kind of blackboard object. For example, given a specific knowledge source, it might use forward-chaining on one kind of `Blackboard` object and backward-chaining on another. Furthermore, given a specific `Blackboard` object, how it is evaluated will depend on which knowledge source is applied.

10.4 Post-Transition

In this section, we consider an improvement to the functionality of the cryptanalysis system and observe how our design weathers the change.

System Enhancements

In any intelligent system, it is important to know what the final answer is to a problem, but it is often equally important to know how the system arrived at this solution. Thus, we desire our application to be introspective: It should keep track of when knowledge sources were activated, what assumptions were made and why, and so on, so that we can later question it, for example, about why it made an assumption, how it arrived at another assumption, and when a particular knowledge source was activated.

To add this new functionality, we need to do two things. First, we must devise a mechanism for keeping track of the work that the controller and each knowledge source perform, and second, we must modify the appropriate operations so that they record this information. Basically, the design calls for the knowledge sources and the controller to register what they did in some central repository.

Let's start by inventing the classes needed to support this mechanism. First, we might define the class `Action`, which serves to record what a particular knowledge source or controller did. Figure 10–16 presents the design of the `Action` class as it fits into our architectural design.

For example, if the controller selected a particular knowledge source for activation, it would create an instance of this class, set the `who` argument to itself, set the `what` argument to the knowledge source, set the `why` argument to some explanation (perhaps including the current priority of the hint), and set `when` this occurred.

The first part of our task is done, and the second part is almost as easy. Consider for a moment where important events take place in our application. As it turns out, five primary kinds of operations are affected:

1. Methods that state an assumption
2. Methods that retract an assumption
3. Methods that activate a knowledge source
4. Methods that cause rules to be evaluated
5. Methods that register hints from a knowledge source

Actually, these events are largely constrained to two places in the architecture: as part of the controller's finite state machine and as part of the assumption mechanism. Our maintenance task, therefore, involves touching all the methods that play a role in these two places, a task that is tedious but by no means rocket science. Indeed, the most important discovery is that adding this new behavior requires no significant architectural change.

To complete our work here, we must also implement a class that can answer who, what, when, and why questions from the user. The design of such an object is not terribly difficult because all the information it needs to know may be found as the state of instances of the class actions.

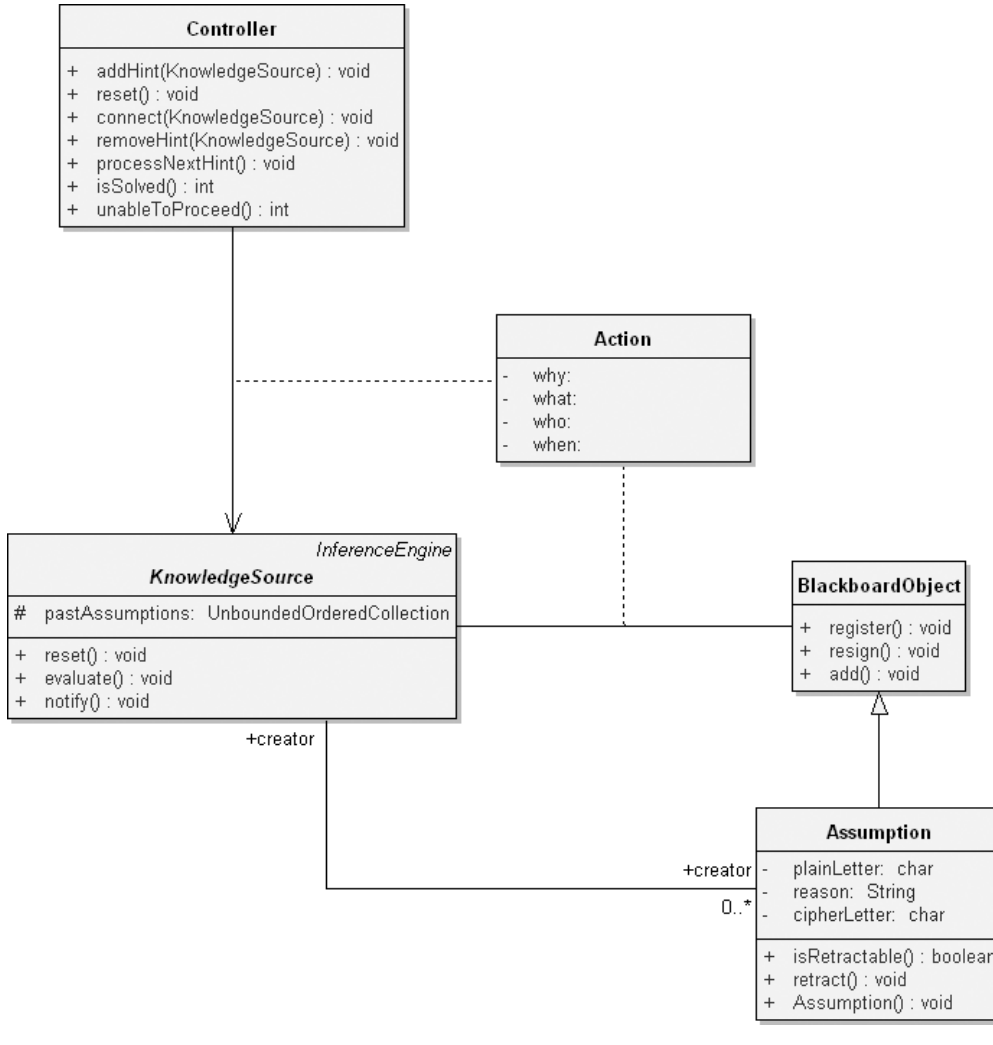

Figure 10–16 Additional Functionality Provided through the `Action` Class Design

Changing the Requirements

Once we have a stable implementation in place, many new requirements can be incorporated with minimal changes to our design. Let's consider three kinds of new requirements:

1. The ability to decipher languages other than English
2. The ability to decipher using transposition ciphers as well as single substitution ciphers
3. The ability to learn from experience

The first change is fairly easy because the fact that our application uses English is largely immaterial to our design. Assuming the same character set is used, it is mainly a matter of changing the rules associated with each knowledge source. Actually, changing the character set is not that difficult either because even the `Alphabet` class is not dependent on what characters it manipulates.

The second change is much harder, but it is still possible in the context of the blackboard framework. Basically, our approach is to add new sources of knowledge that embody information about transposition ciphers. Again, this change does not alter any existing key abstraction or mechanism in our design; rather, it involves the addition of new classes that use existing facilities, such as the `InferenceEngine` class and the assumption mechanism.

The third change is the hardest of all, mainly because machine learning is on the fringes of our knowledge in artificial intelligence. As one approach, when the controller discovers it can no longer proceed, it might ask the user for a hint. By recording this hint, along with the actions that led up to the system being stuck, the blackboard application can avoid a similar problem in the future. We can incorporate this simplistic learning mechanism without vastly altering any of our existing classes; as with all the other changes, this one can build on existing facilities.

Chapter 11

Data Acquisition: Weather Monitoring Station

For many scientific systems, the automatic collection of data is usually acquired through the use of sensors or devices. This data acquisition typically involves the processing of signals and waveforms to obtain the desired information. The components of a data acquisition system include the appropriate sensors that convert any measured parameter to an electrical signal, which is acquired by data acquisition hardware. Control software is developed that interprets the signals for analysis and display.

Using object-oriented techniques to design a data acquisition system allows us to isolate the hardware that measures and collects the data from the application that then analyzes the information. A robust architecture can be defined to allow for sensors and devices to be added or replaced without disturbing the architecture of the control application. Applying interfaces that act as a skin overlaying the hardware allows for isolation of the measuring devices from the application that processes the information. In this chapter, we provide an example of a data acquisition system, in our case, a Weather Monitoring System. The Weather Monitoring System uses sensors and devices that measure the weather conditions that are analyzed and displayed. This example illustrates an object-oriented solution to a real-time control processing application that provides a reusable component architecture that can isolate the hardware from the application.

11.1 Inception

Our Weather Monitoring System is a simple application, encompassing only a handful of classes. Indeed, at first glance, the object-oriented novice might be tempted to tackle this problem in an inherently non-object-oriented manner by considering the flow of data and the various input/output mappings involved. However, as we shall see, even a system as small as this one lends itself well to an object-oriented architecture, and in so doing exposes some of the basic principles of the object-oriented development process.

Requirements for the Weather Monitoring Station

This system shall provide automatic monitoring of various weather conditions. Specifically, it must measure the following:

- Wind speed and direction
- Temperature
- Barometric pressure
- Humidity

The system shall also provide these derived measurements:

- Wind chill
- Dew point temperature
- Temperature trend
- Barometric pressure trend

The system shall have a means of determining the current time and date, so that it can report the highest and lowest values of any of the four primary measurements during the previous 24-hour period.

The system shall have a display that continuously indicates all eight primary and derived measurements, as well as the current time and date. Through the use of a keypad, the user may direct the system to display the 24-hour high or low value of any one primary measurement, together with the time of the reported value.

The system shall allow the user to calibrate its sensors against known values and to set the current time and date.

Defining the Boundaries of the Problem

We begin our analysis by considering the hardware on which our software must execute. This is inherently a problem of systems analysis, involving manufacturability and cost issues that are far beyond the scope of this text. To bound our problem and thus allow us to expose the issues of its software analysis and design, we will make the following strategic assumptions.

- The processor (i.e., CPU) may take the form of a PC or a handheld device.
- Time and date are supplied by a clock.
- Temperature, barometric pressure, and humidity are measured via remote sensors.
- Wind direction and speed are measured from a boom encompassing a wind vane (capable of sensing wind from any of 16 directions) and cups (which advance a counter for each revolution).
- User input is provided through a keypad.
- The display is an off-the-shelf LCD graphic device.
- A timer interrupts the computer every 1/60 second.

Figure 11–1 provides a deployment diagram that illustrates this hardware platform.

We have chosen to throw some hardware at this problem so that we might better focus on the system's software. Obviously, we could require more software by doing less in hardware (e.g., by eliminating some of the hardware for the user input and graphics device), but in this particular application, changing the hardware/software boundary is largely immaterial to our object-oriented architecture. Indeed, one of the characteristics of an object-oriented system is that it tends to speak in the vocabulary of its problem space and so represents a virtual machine that parallels our abstraction of the problem's key entities. Changing the details of the system's hardware impacts only our abstraction of the lower layers of the system.

The details of hardware interfaces can be easily insulated from our software abstractions by wrapping a class around each such interface. For example, we might devise a simple class for accessing the current time and date. We begin by doing a little isolated class analysis, in which we consider what roles and responsibilities this abstraction should encompass.[1] Thus, we might decide that

1. Actually, instead of first setting out to design a new class from scratch, we should start by looking for an existing class that already satisfies our needs. A time and data class is certainly a good candidate for reuse. However, for the purposes of this chapter, we will assume that no such class could be found.

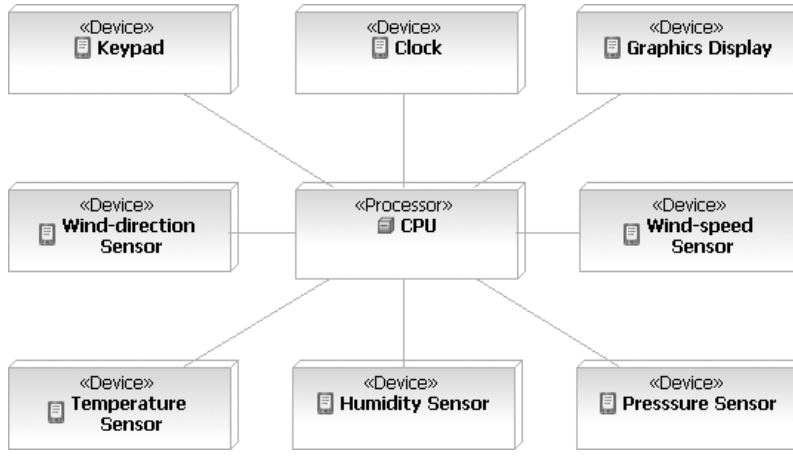

Figure 11–1 The Deployment Diagram for the Weather Monitoring System

this class is responsible for keeping track of the current time in hours, minutes, and seconds, as well as the current month, day, and year. Our analysis might decide to turn these responsibilities into two services, denoted by the operations `currentTime` and `currentDate`, respectively. The operation `currentTime` returns a string in the following format:

```
13:56:42
```

showing the current hour, minute, and second. The operation `currentDate` returns a string in the following format:

```
6-10-93
```

showing the current month, day, and year.

Further analysis suggests that a more complete abstraction would allow a client to chose either a 12- or 24-hour format for the time, which we may provide in the form of an additional modifier named `setFormat`.

By specifying the behavior of this abstraction from the perspective of its public clients, we have devised a clear separation between its interface and implementation. The basic idea here is to build the outside view of each class as if we had complete control over its underlying platform, then implement the class as a bridge to its real inside view. Thus, the implementation of a class at the system's hardware/software boundary serves to bolt the outside view of the abstraction to its underlying platform, which is often constrained by system decisions that are out of the hands of the software engineer. Of course, the gap between an abstrac-

tion's outside and inside views must not be so wide as to require a thick and inefficient implementation to glue the two views together.

One responsibility of our time and date class must therefore include setting the date and time. Carrying out this responsibility requires a new set of services to set the time and date, which we provide via the operations `setHour`, `setMinute`, `setSecond`, `setDay`, `setMonth`, and `setYear`.

We may summarize our abstraction of a time/date class as follows.

Class name:
`TimeDate`
Responsibility:
Keep track of the current time and date.
Operations:
`currentTime`
`currentDate`
`setFormat`
`setHour`
`setMinute`
`setSecond`
`setMonth`
`setDay`
`setYear`
Attributes:
`time`
`date`

Instances of this class have a dynamic lifecycle, which we can express in the state transition diagram shown in Figure 11–2. Here we see that upon initialization, an instance of this class resets its `time` and `date` attributes and then unconditionally enters the `Running` state, where it begins in 24-hour mode. Once in the `Running` state, receipt of the operation `setFormat` toggles the object between 12- and 24-hour mode. No matter what its nested state, however, setting the time or date causes the object to renormalize its attributes. Similarly, requesting its time or date causes the object to calculate a new string value.

We have specified the behavior of this abstraction in enough detail that we can offer it for use in scenarios with other clients we might discover during analysis. Before we consider these scenarios, let's specify the behavior of the other tangible objects in our system.

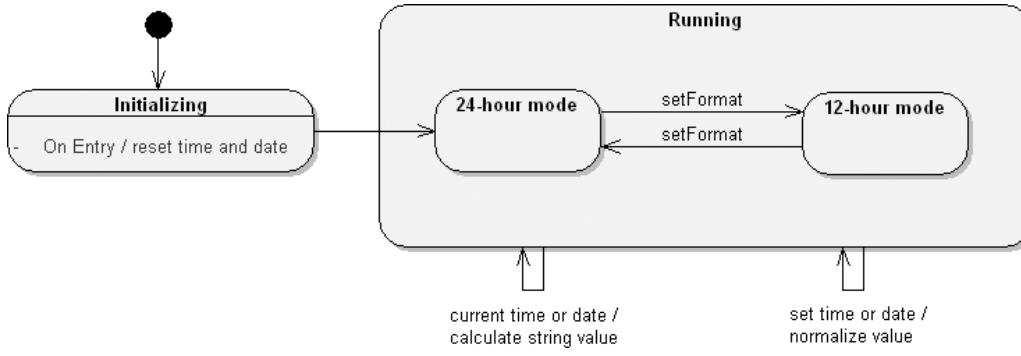

Figure 11–2 The `TimeDate` Lifecycle

The class `Temperature Sensor` serves as an analog to the hardware temperature sensors in our system. Isolated class analysis yields the following first cut at this abstraction's outside view.

Class name:
`Temperature Sensor`
Responsibility:
Keep track of the current temperature.
Operations:
`currentTemperature`
`setLowTemperature`
`setHighTemperature`
Attribute:
`temperature`

The operation `currentTemperature` is self-explanatory. The other two operations derive directly from our requirements, which obligate us to provide a mechanism for calibrating each sensor. For the moment, we will assume that each temperature sensor value is represented by a fixed-point number, whose low and high points can be calibrated to fit known actual values. We translate intermediate numbers to their actual temperatures by simple linear interpolation between these two points, as illustrated in Figure 11–3.

The careful reader may wonder why we have proposed a class for this abstraction, when our requirements imply that there is exactly one temperature sensor in the system. That is indeed true, but in anticipation of reusing this abstraction, we choose to capture it as a class, thereby decoupling it from the particulars of this one system. In fact, the number of temperature sensors monitored by a particular

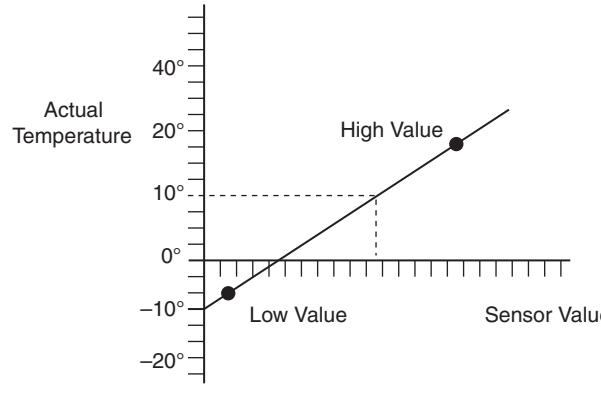

Figure 11–3 `Temperature Sensor` Calibration

system is largely immaterial to our architecture, and by devising a class, we make it simple for other programs in this family of systems to manipulate any number of sensors.

We can express our abstraction of the barometric pressure sensor in the following specification.

Class name:
`Pressure Sensor`
Responsibility:
Keep track of the current barometric pressure.
Operations:
`currentPressure`
`setLowPressure`
`setHighPressure`
Attribute:
`pressure`

A review of the system's requirements reveals that we may have missed one important behavior for this and the previous class, `Temperature Sensor`. Specifically, our requirements compel us to provide a means for reporting the temperature and pressure trends. For the moment (because we are doing analysis, not design), we will be content to focus on the nature of this behavior and, most important, on deciding which abstraction we should make responsible for this behavior.

For both the `Temperature Sensor` and the `Pressure Sensor`, we can express the trends as floating-point numbers between –1 and 1, representing the

slope of a line fitting a number of values over some interval of time.[2] Thus, we may add the following responsibility and its corresponding operation to both of these classes.

Responsibility:

Report the temperature or pressure trend as the slope of a line fitting the past values over the given interval.

Operation:

`trend`

Because this behavior is common to both the `Temperature Sensor` and `Pressure Sensor` classes, our analysis suggests the invention of a common superclass, which we will call `Trend Sensor`, responsible for providing this common behavior.

For completeness, we should point out that there is an alternative view of the world that we might have chosen in our analysis. Our decision was to make this common behavior a responsibility of the sensor class itself. We could have decided to make this behavior a part of some external agent that periodically queried the particular sensor and calculated its trend, but we rejected this approach because it was unnecessarily complex. Our original specification of the `Temperature Sensor` and `Pressure Sensor` classes suggested that each abstraction had sufficient knowledge to carry out this trend-reporting behavior, and by combining responsibilities (albeit in the form of a superclass), we end up with a simple and conceptually cohesive abstraction.

Our abstraction of the humidity sensor can be expressed in the following specification.

Class name:

`Humidity Sensor`

Responsibility:

Keep track of the current humidity, expressed as a percentage of saturation from 0% to 100%.

Operations:

`currentHumidity`
`setLowHumidity`
`setHighHumidity`

2. A value of 0 means that the temperature or pressure is stable. A value of 0.1 denotes a modest rise; a value of –0.3 denotes rapidly declining values. A value approaching –1 or 1 suggests an environmental cataclysm, which is beyond the scope of the scenarios our system is expected to handle properly.

Attribute:
`humidity`

The `Humidity Sensor` has no responsibility for calculating its trend and is therefore not a subclass of `Trend Sensor`.

A review of the system's requirements suggests some behavior common to the classes `Temperature Sensor`, `Pressure Sensor`, and `Humidity Sensor`. In particular, our requirements compel us to provide a means of reporting the highest and lowest values of each of these sensors during a 24-hour period. We defer deciding how to carry out this responsibility because that is an issue of design, not analysis. However, because this behavior is common to all three sensor classes, our analysis suggests the invention of a common superclass, which we call `Historical Sensor`, responsible for providing this common behavior.

Class name:
`Historical Sensor`
Responsibility:
Report the highest and lowest values over a 24-hour period.
Operations:
`highValue`
`lowValue`
`timeOfHighValue`
`timeOfLowValue`

`Humidity Sensor` is a direct subclass of `Historical Sensor`, as is `Trend Sensor`, which serves as an intermediate abstract class, bridging our abstractions of `Historical Sensor` and the concrete classes `Temperature Sensor` and `Pressure Sensor`.

Our abstraction of the wind-speed sensor can be expressed in the following specification.

Class name:
`WindSpeed Sensor`
Responsibility:
Keep track of the current wind speed.
Operations:
`currentSpeed`
`setLowSpeed`
`setHighSpeed`
Attribute:
`speed`

Our requirements suggest that we cannot detect the current wind speed directly; rather, we must calculate its value by taking the number of revolutions of the cups on the boom, dividing by the interval over which those revolutions were counted, and then applying a scaling value appropriate to the particular boom assembly. Needless to say, this calculation is one of the secrets of this class; clients could care less how `currentSpeed` is calculated, as long as this operation satisfies its contract and delivers meaningful values.

A quick domain analysis of the last four concrete classes (`Temperature Sensor`, `Pressure Sensor`, `Humidity Sensor`, and `WindSpeed Sensor`) reveals yet another behavior in common: Each of these classes knows how to calibrate itself by providing a linear interpolation against two known data points. Rather than replicating this behavior in all four classes, we instead choose to make this behavior the responsibility of an even higher superclass, which we call `Calibrating Sensor`, whose specification includes the following.

Class name:
`Calibrating Sensor`
Responsibility:
Provide a linear interpolation of values, given two known data points.
Operations:
`currentValue`
`setHighValue`
`setLowValue`

`Calibrating Sensor` is an immediate superclass of `Historical Sensor`.[3]

Our final concrete sensor for wind direction is a bit different because it requires neither calibration nor history. We may express our abstraction of this entity in the following specification.

Class name:
`WindDirection Sensor`
Responsibility:
Keep track of the current wind direction, in terms of points along a compass rose.
Operation:
`currentDirection`

3. This hierarchy passes our litmus test for inheritance: a `Temperature Sensor` is a kind of `Trend Sensor`, which is also a kind of `Historical Sensor`, which in turn is a kind of `Calibrating Sensor`.

Attribute:
`direction`

To unify our sensor abstractions, we generate the abstract base class `Sensor`, which serves as the immediate superclass to both the classes `WindDirection Sensor` and `Calibrating Sensor`. Figure 11–4 illustrates this complete hierarchy.

Although not part of the sensor hierarchy, our abstraction of the keypad for user input has a simple specification.

Class name:
`Keypad`
Responsibility:
Keep track of the last user input.
Operation:
`lastKeyPress`
Attribute:
`key`

Notice that this class has no knowledge of the meaning of any particular key: Instances of this class know only that one of several keys was pressed. We delegate responsibility for interpreting the meaning of these keys to a different class, which we will identify when we apply these concrete boundary classes to our scenarios.

Our abstraction of an `LCD Device` class serves to insulate our software from the particular hardware we might use. To decouple our software from the particular graphics hardware we might use, our analysis leads us to prototype some common displays for the Weather Monitoring System, and then determine our interface needs.

Figure 11–5 provides such a prototype. Here, we have omitted the required display of wind chill and dew point, as well as such details as how to display the 24-hour high or low value of primary measurements. Nonetheless, some patterns emerge: We need to display only text (in two different sizes and two different styles), circles, and lines (of varying thickness). Additionally, we note that some elements of our display are static (such as the label TEMP), while others are dynamic (such as the wind direction). We choose to display both static and dynamic elements via software. In this manner, we lessen the burden on our hardware by eliminating the need for special labels on the LCD itself, but we require slightly more of our software.

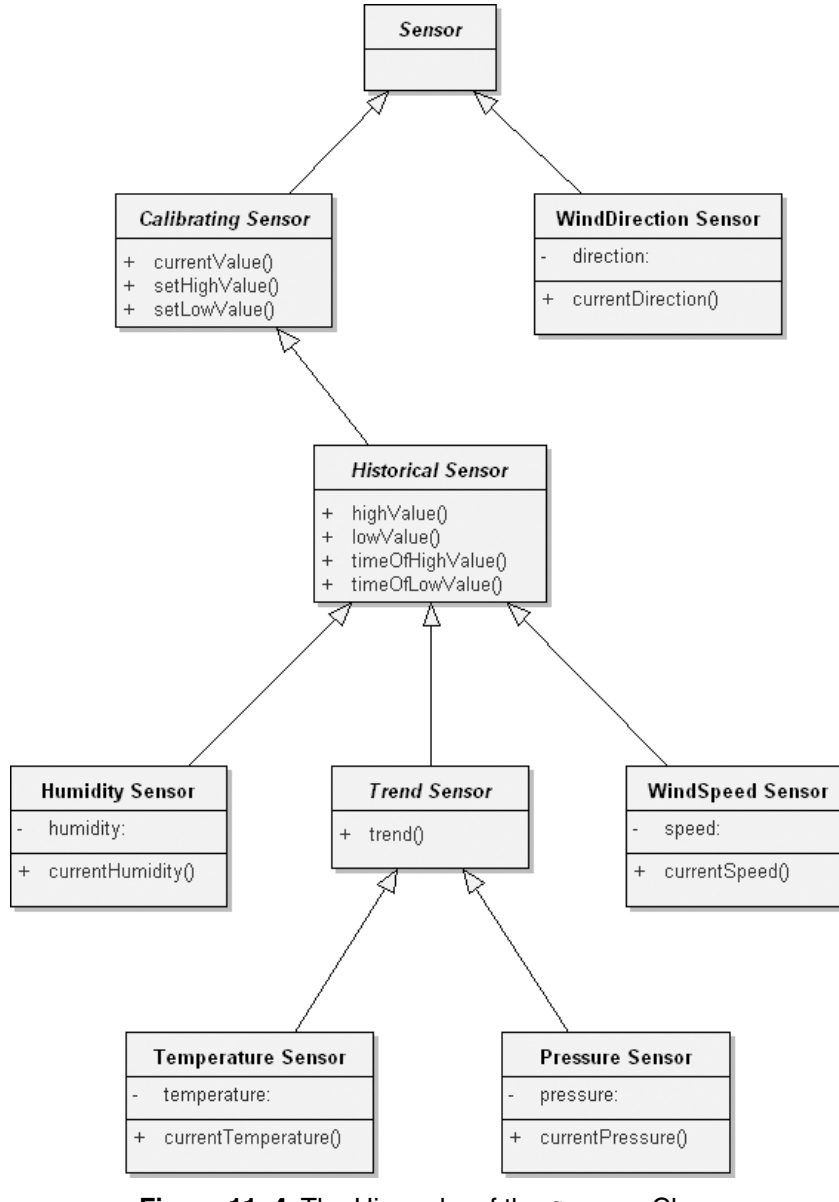

Figure 11–4 The Hierarchy of the `Sensor` Class

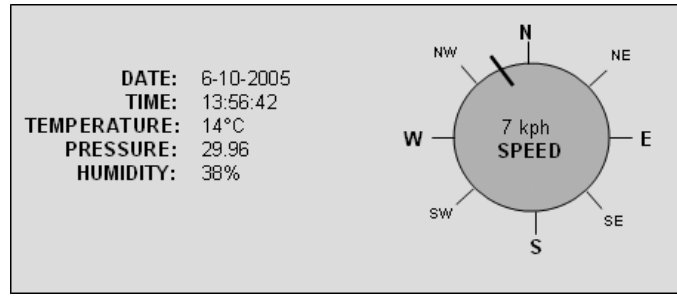

Figure 11–5 The Display for the Weather Monitoring System

We can translate these requirements into the following class specification.

Class name:

`LCD Device`

Responsibility:

Manage the LCD device and provide services for displaying certain graphics elements.

Operations:

```
drawText
drawLine
drawCircle
setTextSize
setTextStyle
setPenSize
```

As with the class `Keypad`, the class `LCD Device` has no knowledge of the meaning of the elements it manipulates. Instances of this class know only how to display text and lines; they do not know what these figures represent. This separation of concerns leaves us with loosely coupled abstractions (which is what we desire), but it does require that we find some agent responsible for mediating between the raw sensors and the display. We defer the invention of this new abstraction until we study some scenarios applicable to this system.

The final boundary class we need to consider is that of the timer. We will make the simplifying assumption that there is exactly one timer per system, whose behavior is to interrupt the computer every 1/60 of a second and, in so doing, to invoke an interrupt service routine. This is a particularly grungy detail, and it would be best if we could hide this implementation detail from the rest of our software abstractions. We can do so by devising a class that uses a callback function and exports only static members (so that we constrain our system to have exactly one timer).

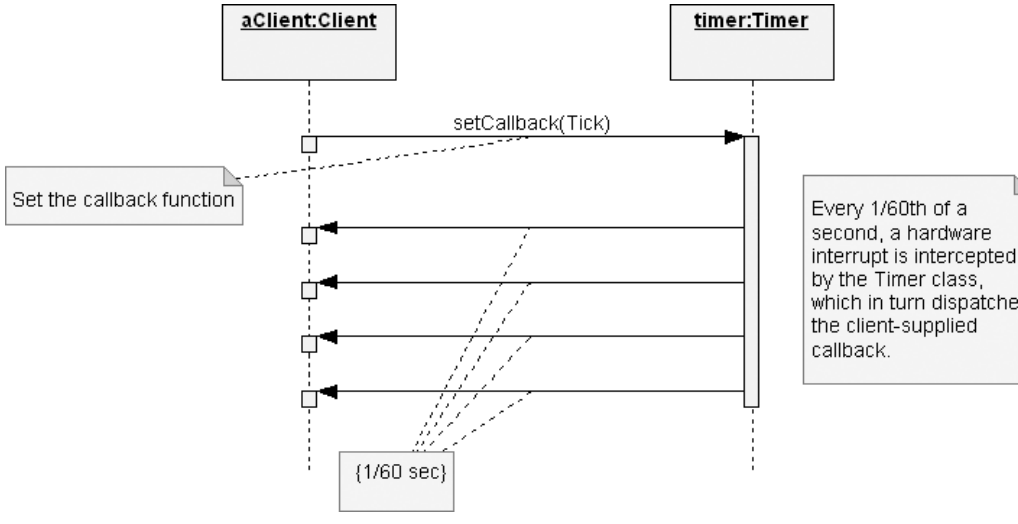

Figure 11–6 The `Timer` Interaction Diagram

Figure 11–6 provides a sequence diagram that illustrates a use case for this abstraction. Here we see how the timer and its client collaborate: The client begins by supplying a callback function, and every 1/60 of a second, the timer calls that function. In this manner, we decouple the client from knowing about how to intercept timed events, and we decouple the timer from knowing what to do when such an event occurs. The primary responsibility that this protocol places on the client is simply that the execution of its callback function must always take less than 1/60 of a second; otherwise, the timer will miss an event.

By intercepting time events, the `Timer` class serves as an active abstraction, meaning that it is at the root of a thread of control. We may express our abstraction of this class in the following specification.

Class name:
`Timer`
Responsibility:
Intercept all timed events and dispatch a callback function accordingly.
Operation:
`setCallback()`

Scenarios

Now that we have established the abstractions at the boundaries of our system, we continue our analysis by studying several scenarios of its use. We begin by enu-

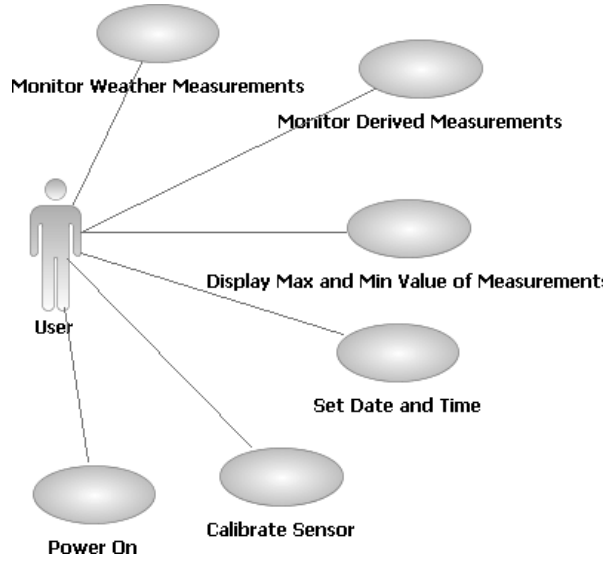

Figure 11–7 Primary Use Cases for the Weather Monitoring System

merating a number of primary use cases (Figure 11–7), as viewed from the point of view of the clients of this system:

- Monitoring basic weather measurements, including wind speed and direction, temperature, barometric pressure, and humidity
- Monitoring derived measurements, including wind chill, dew point, temperature trend, and barometric pressure trend
- Displaying the highest and lowest values of a selected measurement
- Setting the time and date
- Calibrating a selected sensor
- Powering up the system

We add to this list two secondary use cases:

- Power failure
- Sensor failure

11.2 Elaboration

Let's examine a number of these scenarios in order to illuminate the behavior—but not the design—of the system.

Weather Monitoring System Use Cases

Monitoring basic weather measurements is the principal function point of the Weather Monitoring System. One of our system constraints is that we cannot take measurements any faster than 60 times a second. Fortunately, most interesting weather conditions change much more slowly. Our analysis suggests that the following sampling rates are sufficient to capture changing conditions:

- Wind direction: every 0.1 second
- Wind speed: every 0.5 seconds
- Temperature, barometric pressure, and humidity: every 5 minutes

Earlier, we decided that the classes representing each primary sensor should have no responsibility for dealing with timed events. Our analysis therefore requires that we devise an external agent that collaborates with these sensors to carry out this scenario. For the moment, we will defer our specification of the behavior of this agent (how it knows when to initiate a sample is an issue of design, not analysis). The interaction diagram shown in Figure 11–8 illustrates this scenario. Here we see that when the agent begins sampling, it polls each sensor in turn but intentionally skips certain sensors in order to sample them at a slower rate. By polling each sensor rather than letting each sensor act as a thread of control, the execution of our system is more predictable because our agent can control the flow of events. Because this name reflects its place in the behavior of the system, we will make this agent an instance of the class `Sampler`.

We must continue this scenario by asking which of these objects in the interaction diagram is then responsible for displaying the sampled values on the one instance of our `LCD Device` class. Ultimately, we have two choices: We can have each sensor be responsible for displaying itself (the common pattern used in MVC-like architectures), or we can have a separate object be responsible for this behavior. For this particular problem, we choose the latter option because it allows us to

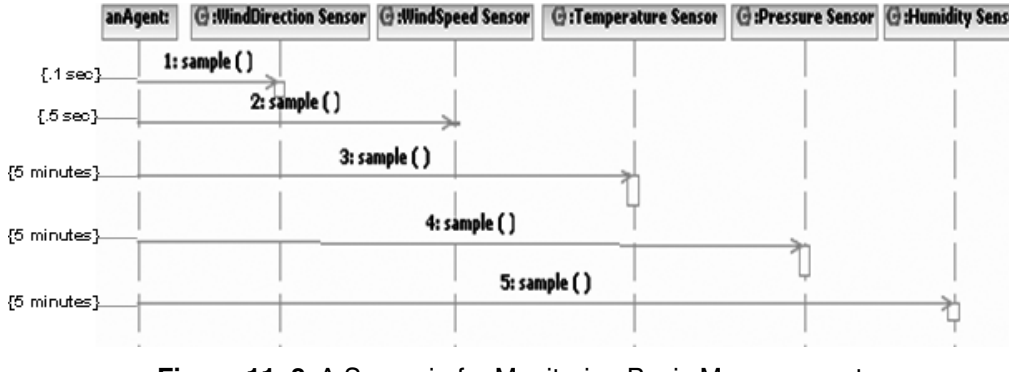

Figure 11–8 A Scenario for Monitoring Basic Measurements

encapsulate all our design decisions about the layout of our display in one class.[4] Thus, we add the following class specification to our products of analysis.

Class name:
`Display Manager`
Responsibility:
Manage the layout of items on the LCD device.
Operations:
`drawStaticItems`
`displayTime`
`displayDate`
`displayTemperature`
`displayHumidity`
`displayPressure`
`displayWindChill`
`displayDewPoint`
`displayWindSpeed`
`displayWindDirection`
`displayHighLow`

The operation `drawStaticItems` exists to draw the unchangeable parts of the display, such as the compass rose used for indicating the wind direction. We will also assume that the operations `displayTemperature` and `displayPressure` are responsible for displaying their corresponding trends (therefore, as we move into implementation, we must provide a suitable signature for these operations).

Figure 11–9 provides a class diagram illustrating the abstractions that must collaborate to carry out this scenario. Note that we also indicate the role that each abstraction plays in its association with other classes.

There is one important side effect from our decision to include the class `Display Manager`.[5] Specifically, internationalizing our software, that is, adapting it to

4. The dominant problem here is where we display each item, not how each item looks. Because this is a decision that is likely to change, it is best for us to encapsulate in one class all the knowledge about where to display each item on the LCD device. Changing our assumptions about front panel layout therefore requires that we touch only one class instead of many.

5. Is this an analysis decision or a design decision? The question can be argued in either direction, although such arguments are largely academic in the face of having to deliver production software. If a decision advances our understanding of the system's desired behavior and in addition leads us to an elegant architecture, we don't really care what it is called.

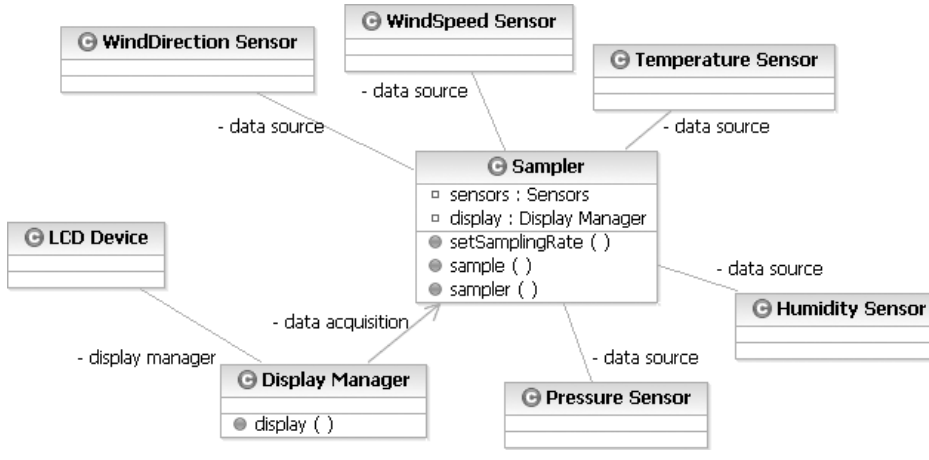

Figure 11–9 The `Sampler` and `Display Manager` Classes

different countries and languages, becomes much easier given this design decision because the knowledge about how elements are named and thus labeled on the display (such as `TEMP` and `WIND`) is part of the secrets of this one class.

Internationalization leads us to consider an issue about which the requirements are silent: Should the system display temperature in Centigrade or Fahrenheit? Similarly, should the system display wind speed in kilometers per hour (kph) or miles per hour (mph)? Ultimately, our software should not constrain us. Because we seek end-user flexibility, we must add an operation `setMode` to both the `Temperature Sensor` and `WindSpeed Sensor` classes. We must also add a new responsibility to each of these classes, which makes their instances construct themselves in a known stable state. Finally, we must modify the signature of the operation `Display Manager::drawStaticItems` accordingly, so that when we change units of measurement, the display manager can update the front panel display if needed.

This discovery leads us to add one more scenario for consideration in our analysis, namely:

- Setting the unit of measurement for temperature and wind speed

We will defer considering this scenario until we study the other use cases that deal with user interaction.

Monitoring the derived measurements for temperature and pressure trends can be achieved through the protocol we have already established for the `Temperature Sensor` and `Pressure Sensor` classes. However, to complete this scenario for all derived measurements, we are now led to discover two new classes, which

we call `Wind Chill` and `Dew Point`, responsible for calculating their respective values. Neither of these abstractions represents sensors because they do not denote any tangible device in the system. Rather, each one acts as an agent that collaborates with two other classes to carry out its responsibilities. Specifically, the `Wind Chill` conspires with the `Temperature Sensor` and `WindSpeed Sensor`, and the `Dew Point` conspires with the `Temperature Sensor` and `Humidity Sensor`. In turn, `Wind Chill` and `Dew Point` collaborate with `Sampler`, using the same mechanism as `Sampler` uses to monitor all the primary weather measurements. Figure 11–10 illustrates the classes involved in this scenario; basically, this class diagram is just a slightly different view of the system than the one shown in Figure 11–9.

Why do we define `Wind Chill` and `Dew Point` as classes, instead of just carrying out their calculation through a simple nonmember function? The answer is that this situation passes our litmus test for object-oriented abstractions: Instances of both `Wind Chill` and `Dew Point` provide some behavior (namely, the calculation of their respective values) and encapsulate some state (each must maintain an association with a particular instance of two different concrete sensors), and each has a unique identity (each particular wind-speed sensor/temperature sensor association must have its own `Wind Chill` object). By "objectifying" these seemingly algorithmic abstractions, we also end up with a more reusable architecture: Both `Wind Chill` and `Dew Point` can be lifted from this particular application because each presents a clear contract to its clients, and each offers a clear separation of concerns relative to all the other abstractions.

Moving on, we next consider the various scenarios that relate to user interaction with the Weather Monitoring System. Deciding on the proper user gestures for

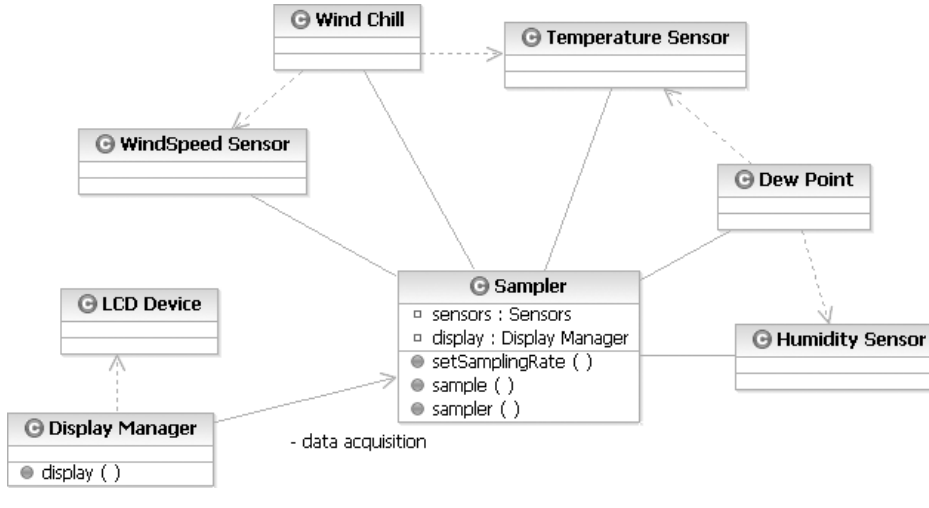

Figure 11–10 Classes for Derived Measurements

interacting with an embedded controller such as this one is still as much of an art as is designing a graphical user interface. A full treatment of how to devise such user interfaces is beyond the scope of this text, but the basic message for the software analyst is that prototyping works and indeed is fundamental in helping to mitigate the risks involved in user interface design. Furthermore, by implementing our decisions in terms of an object-oriented architecture, we make it relatively easy to change these user interface decisions without rending the fabric of our design.

Consider some possible use case scenarios of user interaction.

Use case name:

`Display Max and Min Value of Measurements`

Description:

This use case displays the maximum and minimum values of a selected measurement.

Basic flow:

1. The use case begins when the user presses the SELECT key.
2. The system displays SELECTING.
3. The user presses any one of the keys WIND, TEMP, PRESSURE, or HUMIDITY; any other key press (except RUN) is ignored.
4. The system flashes the corresponding label.
5. The user presses the UP or DOWN key to select display of the highest or lowest 24-hour value, respectively; any other key press (except RUN) is ignored.
6. The system displays the selected value, together with its time of occurrence.
7. Control passes back to step 3 or step 5.

   Note that the user may press the RUN key to commit or abandon the operation, at which time the flashing display, the selected value, and the SELECTING message are removed.

This scenario leads us to enhance the `Display Manager` class by adding both the operations `flashLabel` (which causes the identified label to flash or stop flashing, according to an appropriate operation argument) and `displayMode` (which displays a text message on the LCD device).

Setting the time and date follows a similar scenario.

Use case name:

`Set Date and Time`

Description:

This use case sets the date and time.

Basic flow:

1. The use case begins when the user presses the SELECT key.
2. The system displays SELECTING.
3. The user presses either of the keys TIME or DATE; any other key press (except RUN and the keys listed in step 3 of the previous scenario) is ignored.
4. The system flashes the corresponding label; the display also flashes the first field of the selected item (namely, the hours field for the time and the month field for the date).
5. The user presses the LEFT or RIGHT keys to select another field (selection wraps around); the user presses the UP or DOWN keys to raise or lower the value of the selected field.
6. Control passes back to step 3 or step 5.

   Note that the user may press the RUN key to commit or abandon the operation, at which time the flashing display and the SELECTING message are removed, and the time or date are reset.

Calibrating a particular sensor follows a related pattern of user gestures.

Use case name:

`Calibrate Sensor`

Description:

This use case is used to calibrate the sensors.

Basic flow:

1. The use case begins when the user presses the CALIBRATE key.
2. The system displays CALIBRATING.
3. The user presses any one of the keys WIND, TEMP, PRESSURE, or HUMIDITY; any other key press (except RUN) is ignored.
4. The system flashes the corresponding label.
5. The user presses the UP or DOWN keys to select the high or low calibration point.
6. The display flashes the corresponding value.
7. The user presses the UP or DOWN keys to adjust the selected value.
8. Control passes back to step 3 or step 5.

   Note that the user may press the RUN key to commit or abandon the operation, at which time the flashing display and the CALIBRATING message are removed, and the calibration function is reset.

While calibrating, instances of the `Sampler` class must be told to not sample the selected item; otherwise, erroneous information would be displayed to the user. This scenario therefore requires that we introduce two new operations for the

`Sampler` class, namely, `inhibitSample` and `resumeSample`, both of which have a signature that specifies a particular measurement.

Our last primary scenario involving the user interface concerns setting units of measurement.

Use case name:

`Set Unit of Measurement`

Description:

This use case sets the unit of measurement for temperature and wind speed.

Basic flow:

1. The use case begins when the user presses the MODE key.
2. The system displays MODE.
3. The user presses either of the keys WIND or TEMP; any other key press (except RUN) is ignored.
4. The system flashes the corresponding label.
5. The user presses the UP or DOWN keys to toggle the current unit of measurement.
6. The system updates the unit of measurement for the selected item.
7. Control passes back to step 3 or step 5.

   Note that the user may press the RUN key to commit or abandon the operation, at which time the flashing display and the MODE message are removed, and the current unit of measurement for the item is set.

A study of these scenarios leads us to decide on an arrangement for buttons on the keypad (a system decision), which we illustrate in Figure 11–11.

Each of these user interface scenarios involves some form of modality or event-ordered behavior and so is well suited to expression through the use of state transition diagrams. Because these scenarios are so tightly coupled, we choose to devise a new class, `InputManager`, which is responsible for carrying out the following contractual specification.

Figure 11–11 The User Keypad for the Weather Monitoring System

Class name:
`InputManager`
Responsibility:
Manage and dispatch user input.
Operation:
`processKeyPress`

The sole operation, `processKeyPress`, animates the state machine that lives behind instances of this class.

As shown in Figure 11–12, the outermost state machine diagram for this class encompasses four states: `Running`, `Calibrating`, `Selecting`, and `Mode`. These states correspond directly to the earlier scenarios. We transition to the respective states based on the first key press intercepted while `Running`, and we return to the `Running` state when the last key press is again `Run`. Each time we enter `Running`, we clear the message on the display.

We have expanded the `Mode` state to show how we might more formally express the dynamic semantics of our scenario. As we first enter this state, our entry action is to display an appropriate message on the display. We begin in the `Waiting` state and transition out of this state if we intercept a user key press of the `TEMP` or `WIND` keys, which causes us to enter a nested state of `Processing`,

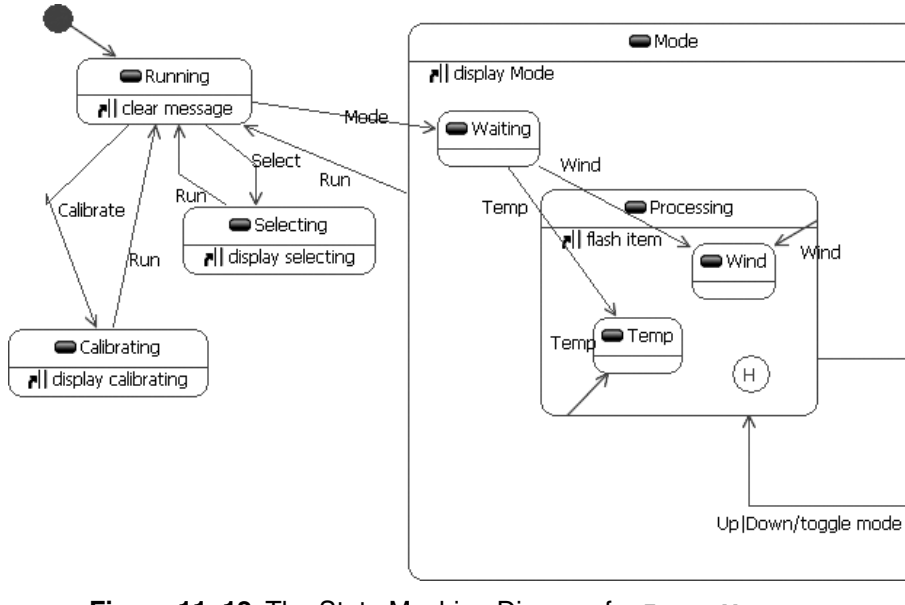

Figure 11–12 The State Machine Diagram for `InputManager`

or a user key press of RUN, which transitions us back to the outermost Running state. Each time we enter Processing, we flash the appropriate item; in subsequent entries to this state, we enter the previously entered nested state, Temp or Wind.

While in the Temp or Wind state, we may intercept one of five key presses: UP or DOWN (which toggles the corresponding mode), TEMP or WIND (which reenters the appropriate nested state), or RUN (which ejects us from the outer Mode state).

The Selecting and Calibrating states similarly expand out to reveal more nested states. We will not show their expanded state machine diagrams here because their presentation does not reveal anything particularly interesting about the problem at hand.[6]

Our final primary scenario involves powering up the system, which requires that we bring all of its objects to life in an orderly fashion, ensuring that each one starts in a stable initial state. We may write a script for our analysis of this scenario as follows.

Use case name:

Power On

Description:

Power up the system.

Basic flow:

1. This use case begins when power is applied.
2. Each sensor is constructed; historical sensors clear their history, and trend sensors prime their slope-calculating algorithms.
3. The user input buffer is initialized, causing garbage key presses (due to noise on power up) to be discarded.
4. The static elements of the display are drawn.
5. The sampling process is initiated.

Postconditions:

The past high/low values of each primary measurement is set to the value and time of their first sample.

The temperature and pressure trends are flat.

The InputManager is in the Running state.

6. Of course, for a production product, a comprehensive analysis would complete the exposition of this state transition diagram. We can defer this task here because it is more tedious than not and in fact does not reveal anything we do not already know about the system under construction.

Notice the use of postconditions in our script to specify the expected state of the system after this scenario completes. As we shall see, there is no one agent in the system that carries out this scenario; rather, this behavior results from the collaboration of a number of objects, each of which is given the responsibility to bring itself to a stable initial state.

This completes our study of the Weather Monitoring System's primary scenarios. To be utterly complete, we might want to walk through the various secondary scenarios. At this point, however, we have exposed a sufficient number of the system's function points, and we want to proceed with architectural design, so that we might begin to validate our strategic decisions.

Every software system needs to have a simple yet powerful organizational philosophy (think of it as the software equivalent of a sound bite that describes the system's architecture), and the Weather Monitoring System is no exception. The next step in our development process is to articulate this architectural framework, so that we might have a stable foundation on which to evolve the system's function points.

The Architecture Framework

In data acquisition and process control domains, we might follow many possible architectural patterns, but the two most common alternatives involve either the synchronization of autonomous actors or time-frame-based processing.

In the first pattern, our architecture encompasses a number of relatively independent objects, each of which serves as a thread of control. For example, we might invent several new sensor objects that build on more primitive hardware/software abstractions, with each such object responsible for taking its own sample and reporting back to some central agent that processes these samples. This architecture has its merits; it may be the only meaningful framework if we have a distributed system in which we must collect samples from many remote locations. This architecture also allows for more local optimization of the sampling process (each sampling actor has the knowledge to adjust itself to changing conditions, perhaps by increasing or decreasing its sampling rate as conditions warrant).

However, this architectural pattern is generally not well suited to hard real-time systems, wherein we must have complete predictability over when events take place. Although the Weather Monitoring System is not hard real-time, it does require some modicum of predictable, ordered behavior. For this reason, we turn to an alternative pattern, that of time-frame-based processing.

As we illustrate in Figure 11–13, this model takes time and divides it into several (usually fixed-length) frames, which we further divide into subframes, each of

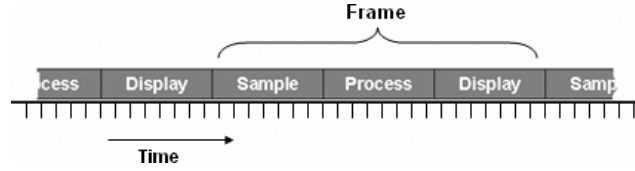

Figure 11–13 Time-Frame Processing

which encompasses some functional behavior. The activity from one frame to another may be different. For example, we might sample the wind direction every 10 frames but sample the wind speed only every 30 frames.[7] The primary merit of this architectural pattern is that we can more rigorously control the order of events.

Figure 11–14 provides a class diagram that expresses this architecture for the Weather Monitoring System. Here we find most of the classes we discovered earlier during analysis, the main difference being that we now show how all the key abstractions collaborate with one another. As is typical in class diagrams for production systems, we do not (and cannot) show every class and every relationship. For example, we have omitted the class hierarchy regarding all of the sensors.

We have invented one new class in this architecture, namely, the class `Sensors`, whose responsibility is to serve as the collection of all the physical sensors in the system. Because at least two other agents in the system (`Sampler` and `InputManager`) must associate with the entire collection of sensors, bundling them in one container class allows us to treat our system's sensors as a logical whole.

11.3 Construction

The central behavior of this architecture is carried out by a collaboration of the `Sampler` and `Timer` classes. We would be wise during architectural design to concretely prototype these classes so that we can validate our assumptions.

The Frame Mechanism

We begin by refining the interface of the class `Timer`, which dispatches a callback function. Figure 11–15 shows the class design.

7. For example, if each frame is allocated to be 1/60 second, 30 frames represents 0.5 second.

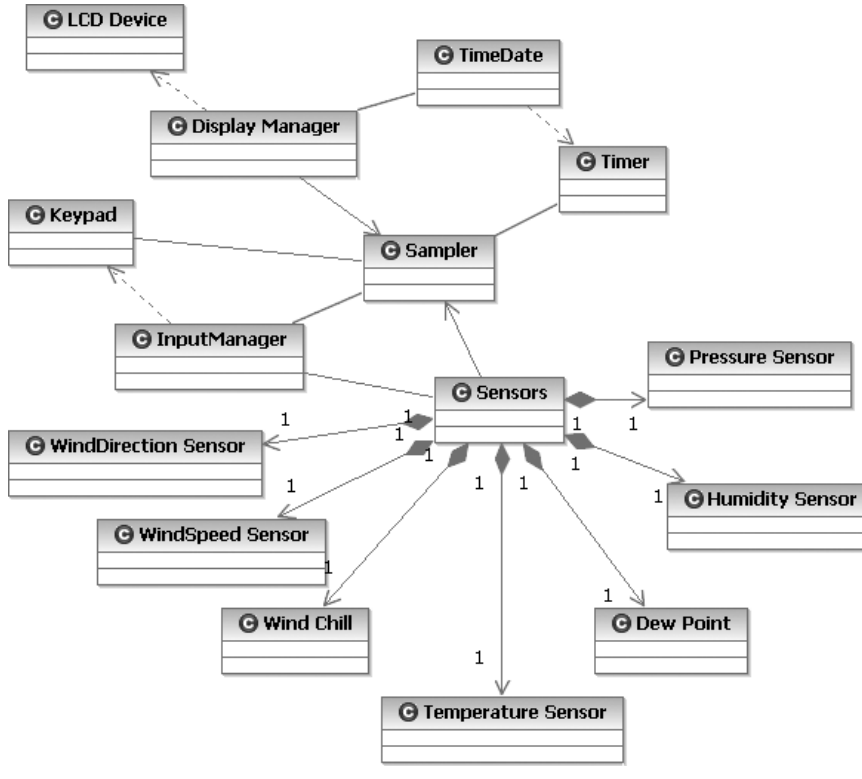

Figure 11–14 The Architecture of the Weather Monitoring System

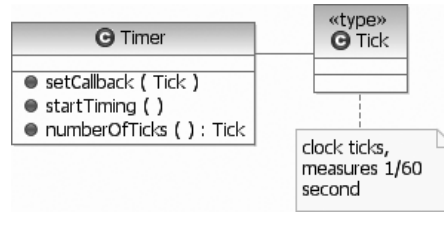

Figure 11–15 The Design of the `Timer` Class

`Timer` is an unusual class, but remember that it holds some unusual secrets. We use the first operation `setCallback` to attach a callback function to the timer. We launch the timer's behavior by invoking `startTiming`, after which time the one `Timer` entity dispatches the callback function every 1/60 of a second. Notice that we introduce an explicit starting operation because we cannot rely on any particular implementation-dependent ordering in the elaboration of declarations.

Before we turn to the `Sampler` class, we introduce a new declaration that names the various sensors in this particular system. The enumeration class

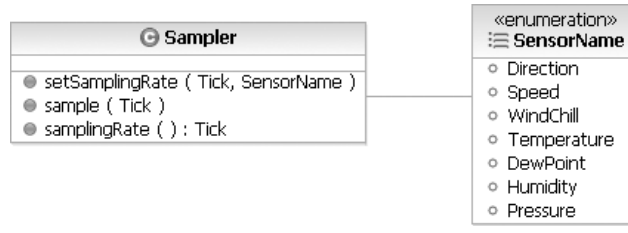

Figure 11–16 The Interface of the `Sampler` Class

`SensorName` contains enumeration literals for all the sensors in our system. Figure 11–16 shows the interface of the `Sampler` class.

We have introduced the modifier `setSamplingRate` and its selector `samplingRate` so that clients can dynamically alter the behavior of the sampling objects.

To tie the `Timer` and `Sampler` classes together, we just need a little bit of C++ glue code. First we declare an instance of `Sampler` and a nonmember function.

```
Sampler sampler;

void acquire(Tick t)
{
  sampler.sample(t);
}
```

Now we can write a fragment of our main function, which simply attaches the callback function to the timer and starts the sampling process.

```
main() {

  Timer::setCallback(acquire);
  Timer::startTiming();

  while(1) {
    ;
  }

  return 0;

}
```

This is a fairly typical main program for object-oriented systems: It is short (because the real work is delegated to key objects in the system), and it involves a

dispatch loop (which in this case does nothing because we have no background processing to complete).[8]

To continue this thread of the system's architecture, we next provide an interface for the `Sensors` class (Figure 11–17). For the moment, we assume the existence of the various concrete sensor classes.

This is basically a collection type class, and for this reason we make `Sensors` a subclass of the foundation class `Collection`.[9] We make `Collection` a protected superclass because we don't want to expose most of its operations to clients of the `Sensors` class. Our declaration of `Sensors` provides only a sparse set of operations because our problem is sufficiently constrained that we know sensors are only added and never removed from the collection.

We have invented a generalized sensor collection class that can hold multiple instances of the same kind of sensor, with each instance within its class distinguished by a unique ID, numbered starting at zero.

We specify in the `Sampler` class the associations with the `Sensors` and `Display Manager` classes and revise our declaration of the one instance of the `Sampler` class (Figure 11–18).

The construction of the `Sampler` object connects this agent with the specific collection of sensors and the particular display manager used in the system.

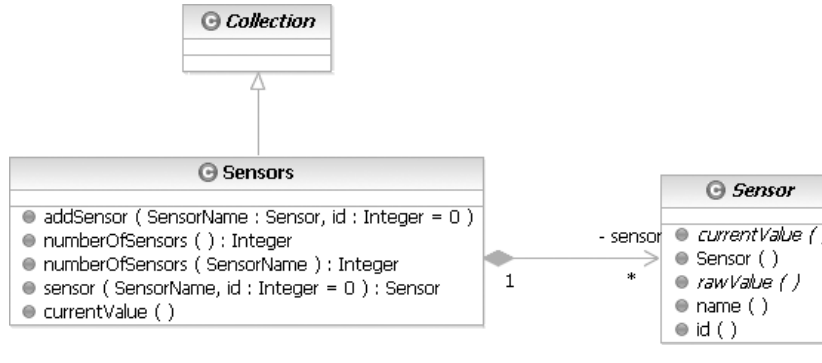

Figure 11–17 The Interface of the `Sensors` class

8. This is yet another common architectural pattern: Dispatch loops serve to intercept external or internal events and then dispatch them to the appropriate agents.

9. The `Collection` class is an abstract superclass that provides common operations for a collection of items supplied by language libraries.

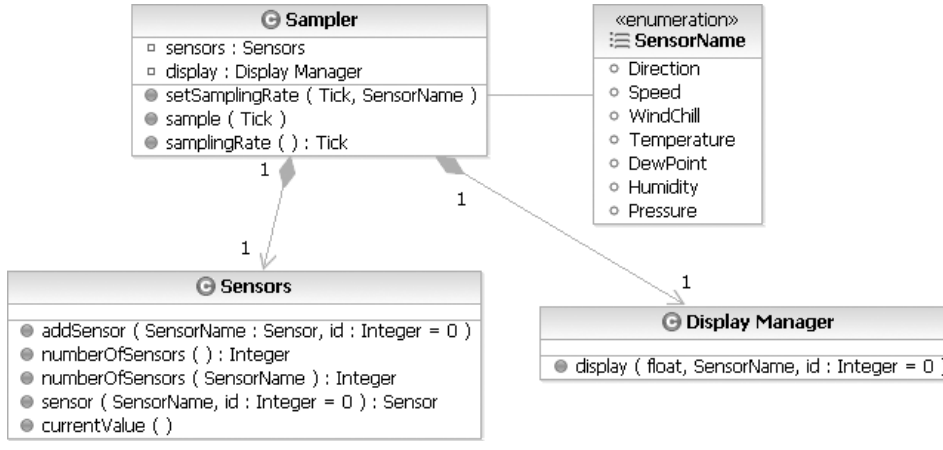

Figure 11–18 The Design of the `Sampler` Class

Now we can implement the `Sampler` class's key operation, `sample`.

```
void Sampler::sample(Tick t)
{
  for (SensorName name = Direction; name <= Pressure; name++)
    for (unsigned int id = 0; id  <
            repSensors.numberOfSensors(name); id++)

      if (!(t % samplingRate(name)))

        repDisplayManager.display(repSensors.sensor(name,
            id).currentValue(), name, id);
}
```

The action of this member function is to iterate through each kind of sensor and, in turn, each unique sensor of that kind in the collection. For each sensor it encounters, `sample` checks to see whether it is time to sample its value and, if so, references the sensor from the collection, takes its current value, and delivers this value to the display manager associated with the `Sampler` instance.[10]

The semantics of this operation relies on the polymorphic behavior of the operation `currentValue` defined for the base class `Sensor`. This operation also relies on the operation `display` defined for the class `Display Manager`.

10. An alternate approach would be to have each sensor provide a member function that returns its sampling rate and another member function that draws the sensor on the LCD. This design would make the implementation of the `Sampler` class simpler and more extensible, although it would shift more responsibilities to the sensor classes.

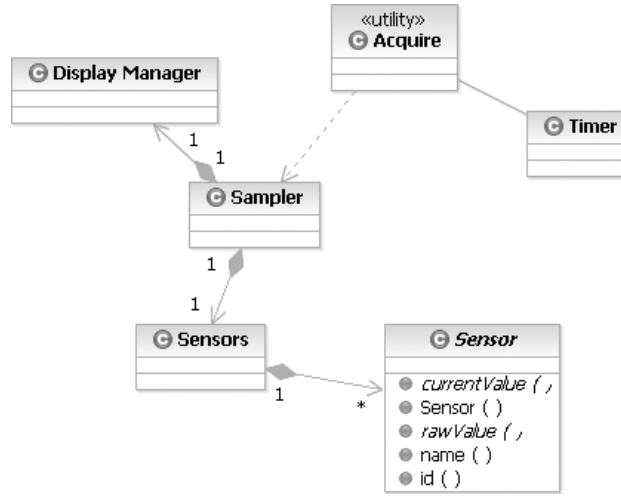

Figure 11–19 The Frame Mechanism

Now that we have refined this element of our architecture, we present a new class diagram in Figure 11–19 that highlights this frame mechanism.

Now that we have validated our architecture by walking through several scenarios, we can continue with the incremental development of the system's function points.

Release Planning

We start this process by proposing a sequence of releases, each of which builds on the previous release.

- Develop a minimal functionality release, which monitors just one sensor.
- Complete the sensor hierarchy.
- Complete the classes responsible for managing the display.
- Complete the classes responsible for managing the user interface.

We could order these releases in just about any manner, but we choose this one, which progresses from highest to lowest risk, thereby forcing our development process to directly attack the hard problems first.

Developing the minimal functionality release forces us to take a vertical slice through our architecture and implement small parts of just about every key abstraction. This activity addresses the highest risk in the project, namely,

whether we have the right abstractions with the right roles and responsibilities. This activity also gives us early feedback because we can now play with an executable system. Forcing early closure like this has a number of technical and social benefits. On the technical side, it forces us to begin to bolt the hardware and software parts of our system together, thereby identifying any impedance mismatches early. On the social side, it allows us to get early feedback about the look and feel of the system, from the perspectives of real users.

Because completing this release is largely a manner of tactical implementation, we will not bother with exposing any more of its structure. We will now turn to elements of later releases because they reveal some interesting insights about the development process.

The Sensor Mechanism

In inventing the architecture for this system, we have already seen how we had to iteratively and incrementally evolve our abstraction of the sensor classes, which we began during analysis. In this evolutionary release, we expect to build on the earlier completion of a minimal functional system and finish the details of this class hierarchy.

At this point in our development cycle, the class hierarchy we first presented in Figure 11–4 remains stable, although, not surprisingly, we had to adjust the location of certain polymorphic operations in order to extract greater commonality. Specifically, in an earlier section we noted the requirement for the `currentValue` operation, declared in the abstract base class `Sensor`. We may complete our design of the `Sensor` class (Figure 11–20).

Notice that through the class constructor, we gave the instances of this class knowledge of their name and ID. This is essentially a kind of runtime type identification, but providing this information is unavoidable here because, per the requirements, each sensor instance must have a mapping to a particular interface. We can hide the secrets of this mapping by making this interface a function of a sensor name and ID.

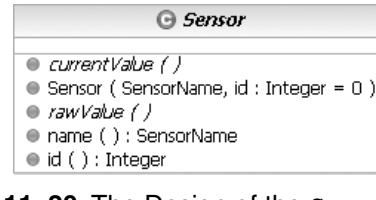

Figure 11–20 The Design of the `Sensor` Class

Now that we have added this new responsibility, we can go back and simplify the signature of `DisplayManager::display` to take only a single argument, namely, a reference to a `Sensor` object. We can eliminate the other arguments to this function because the `Display Manager` can now ask the `Sensor` object its name and ID.

Making this change is advisable because it simplifies certain cross-class interfaces. Indeed, if we fail to keep up with small, rippling changes such as this one, our architecture will eventually suffer as the protocols among collaborating classes become inconsistently applied.

The declaration of the immediate subclass `Calibrating Sensor` builds on the base class `Sensor` (Figure 11–21).

`Calibrating Sensor` introduces two new operations (`setHighValue` and `setLowValue`) and implements the previously defined function `currentValue`.

Next, consider the declaration of the subclass `Historical Sensor`, which builds on the class `Calibrating Sensor` (Figure 11–22).

`Historical Sensor` has four operations whose implementation requires collaboration with the `TimeDate` class for the time of the high or low values. Note that `Historical Sensor` is still an abstract class because we have not yet completed the definition of the abstract function `rawValue`, which we defer to be a concrete subclass responsibility.

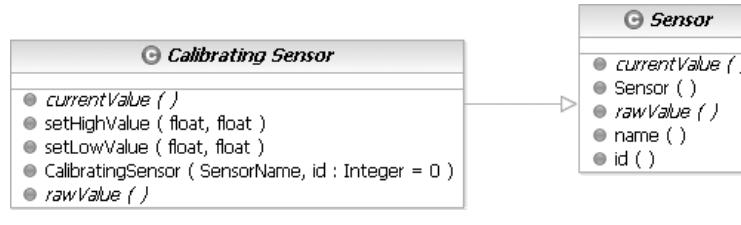

Figure 11–21 The Design of the `Calibrating Sensor` Class

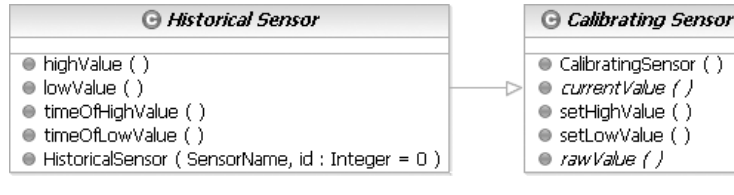

Figure 11–22 The Design of the `Historical Sensor` Class

The class `Trend Sensor` inherits from `Historical Sensor` and adds one new responsibility (Figure 11–23).

`Trend Sensor` introduces one new function. As with some of the other operations that some other intermediate classes have added, we declare `trend` as concrete because we do not desire that subclasses change their behavior.

Ultimately, we reach concrete subclasses such as `Temperature Sensor` (Figure 11–24).

Notice that the signature of this class's constructor is slightly different than its superclass's, simply because at this level of abstraction, we know the specific name of the class. Also, notice that we have introduced the operation `currentTemperature`, which follows from our earlier analysis. This operation is semantically the same as the polymorphic function `currentValue`, but we choose to include both of them because the operation `currentTemperature` is slightly more type-safe.

Once we have successfully completed the implementation of all classes in this hierarchy and integrated them with the previous release, we may proceed to the next level of the system's functionality.

The Display Mechanism

Implementing the next release, which completes the functionality of the classes `DisplayManager` and `LCD Device`, requires virtually no new design work, just some tactical decisions about the signature and semantics of certain func-

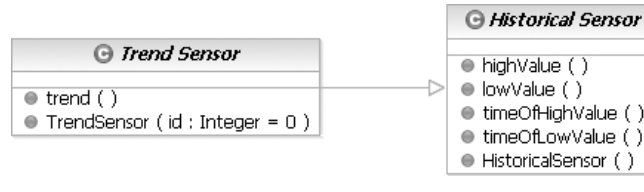

Figure 11–23 The Design of the `Trend Sensor` Class

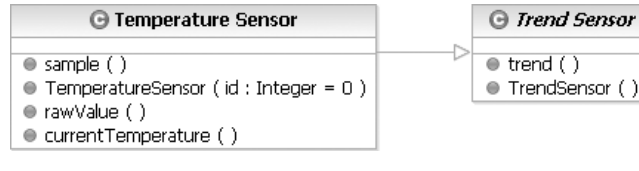

Figure 11–24 The Design of the `Temperature Sensor` Class

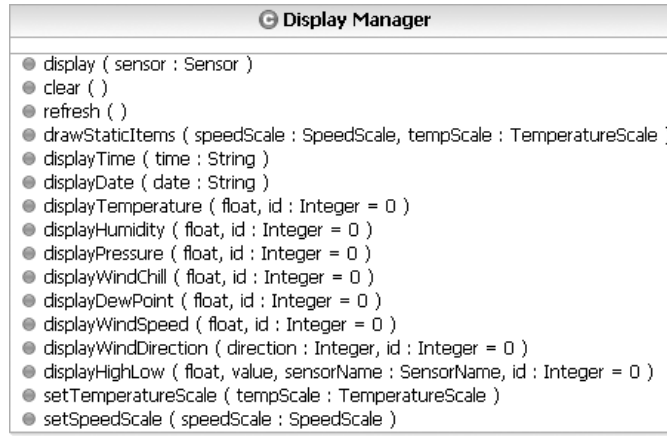

Figure 11–25 The Design of the `Display Manager` Interface

tions. Combining the decisions we made during analysis with our first architectural prototype, wherein we made some important decisions about the protocol for displaying sensor values, we can derive the concrete interface shown in Figure 11–25.

None of these operations are abstract because we neither expect nor desire any subclasses.

Notice that this class exports several primitive operations (such as `displayTime` and `refresh`) but also exposes the composite operation `display`, whose presence greatly simplifies the action of clients that must interact with instances of `Display Manager`.

`Display Manager` ultimately uses the resources of the `LCD Device` class, which, as we described earlier, serves as a skin over the underlying hardware. In this manner, `Display Manager` raises our level of abstraction by providing a protocol that speaks more directly to the nature of the problem space.

The User Interface Mechanism

The focus of our last major release is the tactical design and implementation of the classes `Keypad` and `InputManager`. Similar to the `LCD Device` class, the `Keypad` class serves as a skin over the underlying hardware, which thereby relieves the `InputManager` of the nasty details of talking directly to the hardware. Decoupling these two abstractions also makes it far easier to replace the physical input device without destabilizing our architecture.

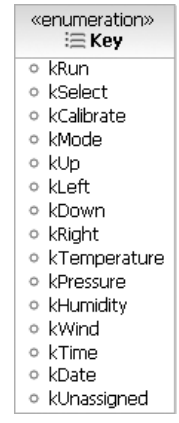

Figure 11–26 The Design of the `Key` Enumeration Class

We start with a declaration that names the physical keys in the vocabulary of our problem space. An enumeration class, `Key`, is defined as shown in Figure 11–26.

We use the `k` prefix to avoid name clashes with literals defined in `SensorName`.

Continuing, we may capture our abstraction of the `Keypad` class as shown in Figure 11–27.

The protocol of this class derives from our earlier analysis. We have added the operation `inputPending` so that clients can query if user input exists that has not yet been processed.

The class `InputManager` has a similarly sparse interface (Figure 11–28).

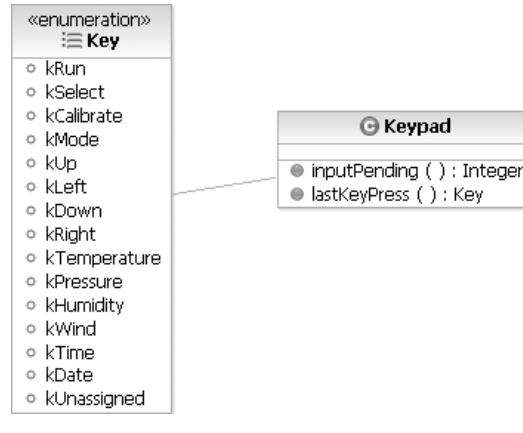

Figure 11–27 The Design of the `Keypad` Class

Figure 11–28 The Design of the `InputManager` Interface

As we will see, most of the interesting work of this class is carried out in the implementation of its finite state machine.

As we illustrated earlier in Figure 11–14, instances of the `Sampler`, `InputManager`, and `Keypad` classes collaborate to respond to user input. To integrate these three abstractions, we must subtly modify the interface of the `Sampler` class to include a new object, `repInputManager` (Figure 11–29).

Through this design decision, we establish an association among instances of the `Sensors`, `Display Manager`, and `InputManager` classes at the time we construct an instance of `Sampler`. This design asserts that instances of `Sampler` must always have a collection of sensors, a display manager, and an input manager.

We must also incrementally modify the implementation of the function `Sampler::sample`.

```
void Sampler::sample(Tick t)
{
  repInputManager.processKeyPress();
  for (SensorName name = Direction; name <= Pressure; name++)
    for (unsigned int id = 0; id <
       repSensors.numberOfSensors(name); id++)
      if (!(t % samplingRate(name)))
        repDisplayManager.display(repSensors.sensor(name,
        id));
}
```

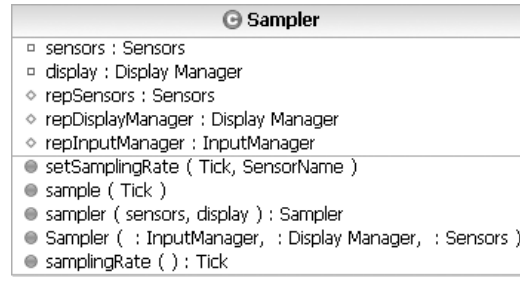

Figure 11–29 The Revised Design of the `Sampler` Interface

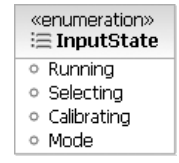

Figure 11–30 The Design of the `InputState` Enumeration Class

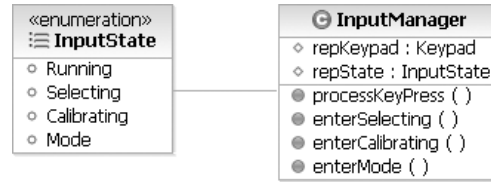

Figure 11–31 `InputManager` with the `InputState` Class Design

Here we have added an invocation to `processKeyPress` at the beginning of every time frame.

The `processKeyPress` operation is the entry point to the finite state machine that drives the instances of this class. Ultimately, there are two approaches we can take to implement this or any other finite state machine: We can explicitly represent states as objects (and thereby depend on their polymorphic behavior), or we can use enumeration literals to denote each distinct state.

For modest-sized finite state machines such as the one embodied by the `InputManager` class, it is sufficient for us to use the latter approach. Thus, we might first introduce the names of the class's outermost states (Figure 11–30).

Next, we introduce some protected helper functions (Figure 11–31).

Finally, we can begin to implement the state transitions we first introduced in Figure 11–12.

```
void InputManager::processKeyPress()
{
  if (repKeypad.inputPending()) {
    Key key = repKeypad.lastKeyPress();
    switch (repState) {
      case Running:
        if (key == kSelect)
          enterSelecting();
        else if (key == kCalibrate)
          enterCalibrating();
        else if (key == kMode)
```

```
            enterMode();
          break;
        case Selecting:
          ...
          break;
        case Calibrating:
           ...
          break;
        case Mode:
           ...
           break;
        }
    }
}
```

The implementation of this function and its associated helper functions thus parallels the state transition diagram shown in Figure 11–12.

11.4 Post-Transition

The complete implementation of this basic Weather Monitoring System is of modest size, encompassing only about 20 classes. However, for any truly useful piece of software, change is inevitable. Let's consider the impact of two enhancements to the architecture of this system.

Our system thus far provides for the monitoring of many interesting weather conditions, but we may soon discover that users want to measure rainfall as well. What is the impact of adding a rain gauge?

Happily, we do not have to radically alter our architecture; we must merely augment it. Using the architectural view of the system from Figure 11–14 as a baseline, to implement this new feature, we must do the following.

- Create a new class, `RainFall Sensor`, and insert it in the proper place in the sensor class hierarchy (a `RainFall Sensor` is a kind of `Historical Sensor`).
- Update the enumeration `SensorName`.
- Update the `Display Manager` so that it knows how to display values of this sensor.
- Update the `InputManager` so that it knows how to evaluate the newly-defined key `RainFall`.
- Properly add instances of this class to the system's `Sensors` collection.

We must deal with a few other small tactical issues needed to graft in this new abstraction, but ultimately, we need not disrupt the system's architecture or its key mechanisms.

Let's consider a totally different kind of functionality. Suppose we desire the ability to download a day's record of weather conditions to a remote computer. To implement this feature, we must make the following changes.

- Create a new class, `SerialPort`, responsible for managing a port used for serial communication.
- Invent a new class, `Report Manager`, responsible for collecting the information required for the download. Basically, this class must use the resources of the collection class `Sensors` together with its associated concrete sensors.
- Modify the implementation of `Sampler::sample` to periodically service the serial port.

It is the mark of a well-engineered object-oriented system that making this change does not rend our existing architecture but, rather, reuses and then augments its existing mechanisms.

Web Application: Vacation Tracking System

For many businesses today, the independence of workers has been ever increasing. It is not uncommon for workers to divide their time across multiple projects and to report to multiple project managers. As a result, managers have fewer informal interactions with their workers and find it increasingly difficult to be aware of and manage their workers' vacation time. Thus, in the example explored in this chapter, our fictitious company has decided to develop and deploy a flexible vacation time management application for managers and employees alike to use to manage their vacation time.

The decision by a large enterprise organization to implement this and similar functionality is never made in isolation. It is unlikely that this proposed system is the first that this organization has ever created. This desired functionality must be considered in the context of any existing systems (and perhaps even with other proposed systems). As a result, many architectural decisions will already be made a priori, and in our situation, the decision to deliver this functionality as a Web application not only is well suited to the task but also will be considered an extension to the organization's existing intranet, thus providing a convenient and natural entry point for its users.

In the following sections, we discuss a summary of the architecturally significant and interesting aspects of this system. Given the space limitations of this book, we express only those aspects and content necessary to impart a general feel and understanding of the application. In the real world, an accurate and complete discussion of such a system would require significantly more content, models, and examples. However, here we cover the key or central principles of object orientation as they apply to

this sort of Web application development, and we discuss several interesting issues unique to the Web application development process.

12.1 Inception

The system's requirements are presented by a summary of the vision document, key features, the use case model, key use case specifications, and architecturally significant line item requirements.

The Requirements

The vision for this project can be summarized easily.

> A Vacation Tracking System (VTS) will provide individual employees with the capability to manage their own vacation time, sick leave, and personal time off, without having to be an expert in company policy or the local facility's leave policies.

The most important goal of this system is to give individual employees the capability and responsibility to manage this particular aspect of their employment agreements with the company. The underlying motivations for this desire include the need to streamline the functions of the human resources (HR) department, to minimize noncore, business-related activities of management, and to give a sense of empowerment to the employees. These objectives will be met only if the system developed is easy to use, intuitive, and intelligent. An overriding design goal can therefore be stated simply.

> The system must be easy to use.

Anyone reading this with even the most minimal experience developing commercial software as part of a team effort will be rolling their eyes at such a vague requirement. Something so blatantly general and subjective should never be recorded as an official requirement, right? Not necessarily so. While it is clear that this simple line item is not a hard and traceable requirement, it does represent an honest feature of the desired system. It is so important that unless it at least appears to have been met by the developers and ultimately accepted by the end users alike, the delivered application will fail.

High-level and potentially vague features like this sometimes need to be part of the official requirements set, not so they can be objectively verified and tracked through testing, but rather so they can be used to support and justify other, more concrete requirements or design decisions. As an example, consider the require-

ments for a Web application that at some point requires the end user to submit to the system some formatted text (e.g., bold, italics, lists, paragraphs, and so on). The architects have two potential solutions. They can require the end user to incorporate special codes into the body of the text, as many popular bulletin board systems do today, or they can use a custom-built or commercially available Java applet that provides WYSIWYG formatting. The technical complexity and risk of such an applet are significant, yet the end-user advantages are also equally clear. If one of the primary system features or design goals was ease of use, the use of this applet could be justified. However, if ease of use was simply a desired rather than critical requirement, introducing the potential complexity and risk of the applet could not be justified.

The main goal of this application is to improve the internal business processes of this organization, at least with respect to the time it takes to manage vacation time requests. In the past, all vacation time had to be approved by an immediate manager and then checked by a clerk in the HR department before it was authorized. Sometimes this manual process could take days. An automated system will speed up this process and will require at most one manual approval by the immediate manager (some high-level employees may not require manager approval).

This system has the potential to save time and money mostly in the HR department, which is essentially taken out of the individual time request process and replaced by a rules-based validation system. HR personnel are still responsible for entering and updating employee vacation data in the system; however, they will no longer be a link in the chain for requesting and validating each time request.

The system will provide the following key features:

- Implements a flexible rules-based system for validating and verifying leave time requests
- Enables manager approval (optional)
- Provides access to requests for the previous calendar year, and allows requests to be made up to a year and a half in the future
- Uses e-mail notification to request manager approval and notify employees of request status changes
- Uses existing hardware and middleware
- Is implemented as an extension to the existing intranet portal system, and uses the portal's single-sign-on mechanisms for all authentication
- Keeps activity logs for all transactions
- Enables the HR and system administration personnel to override all actions restricted by rules, with logging of those overrides
- Allows managers to directly award personal leave time (with system-set limits)

- Provides a Web service interface for other internal systems to query any given employee's vacation request summary
- Interfaces with the HR department legacy systems to retrieve required employee information and changes

The Use Case Model

The top-level use case model contains four actors and eight use cases, as shown in Figure 12–1. The granularity of the use cases is reasonably coarse. For example, the use case `Manage Time` describes functionality, invoked by the `Employee`, that includes viewing, creating, and canceling vacation time requests. A use case is not a description of a single functional requirement but rather a description of something that provides significant value to the actor invoking it, in the form of scenarios. Just being able to view a vacation time request, for example, provides minimal value, but being able to manage your own vacation time does provide significant value.

An interesting observation about use cases of Web-centric systems is that they tend to be expressed very strictly in terms of stimulus and response. That is, a use case scenario is typically expressed as a list of actor actions and immediate system

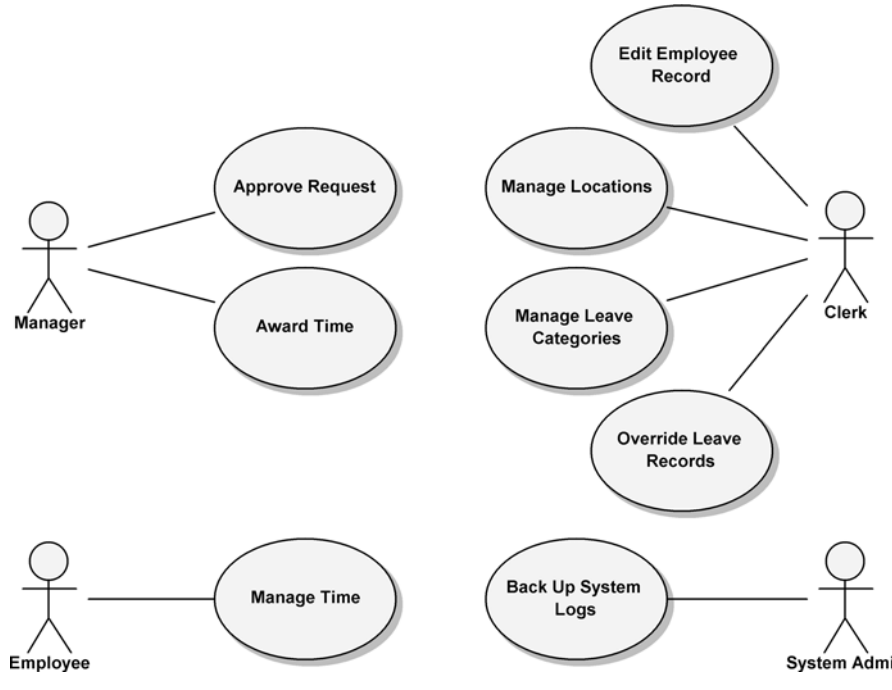

Figure 12–1 The Top-Level Use Case Model

responses. Also, there is typically a high degree of correlation of system responses to actual and individually named Web pages or screens. This is in contrast to non-Web applications, where content in the scenarios of activity focuses more on the information and use gestures being exchanged than on the identification of discrete user interface units, such as Web pages in a Web-centric system.

The system contains the following actors.

- `Employee`: The main user of this system. An employee uses this system to manage his or her vacation time.
- `Manager`: An employee who has all the abilities and goals of a regular employee, but with the added responsibility of approving vacation requests for immediate subordinates. A manager may award subordinates comp time, subject to certain limits set in the system.
- `Clerk`: A member of the HR department who has sufficient rights to view employees' personal data and is responsible for ensuring that employees' information in all HR systems is up to date and correct. An HR clerk can add or remove nearly any record in the system. In the real world, HR clerks may or may not be employees; however, if they are employees, they use two separate login IDs to manage these two different roles.
- `System Admin`: A role responsible for the smooth running of the system's technical resources (e.g., Web server, database) and for collecting and archiving all log files.

The main use cases are as follows.

- `Manage Time`: Describes how employees request and view vacation time requests.
- `Approve Request`: Describes how a manager responds to a subordinate's request for vacation time.
- `Award Time`: Describes how a manager can award a subordinate extra leave time (comp time).
- `Edit Employee Record`: Describes how an HR clerk edits an employee's information in the system. This includes setting all the leave time allowances and the maximum time that can be awarded by the manager.
- `Manage Locations`: Describes how an HR clerk manages location records and their rules.
- `Manage Leave Categories`: Describes how an HR clerk manages leave categories and their rules.
- `Override Leave Records`: Describes how an HR clerk may override any rejection of leave time requests made by the rules in the system.
- `Back Up System Logs`: Describes how the system administrator backs up the system's logs.

12.2 Elaboration

Sometimes it is not always clear when analysis starts or when requirements gathering and understanding during the Inception phase end. This is also why iterative development processes are so popular and the practicality of the waterfall process so often questioned. It is important, however, to have the most important and architecturally significant use cases described and discussed first. All the details need not be complete, but the architecturally significant ones should be addressed before a particular use case can undergo refinement.

In the ideal world, analysis of a system should be independent of an implementing architecture. The reality, however, is that prior knowledge of the implementing architecture may contribute to the shape of the analysis.

This can be especially true when the application under development is a Web application because in most cases the decision to adopt a Web-centric architecture is something that is known at Inception. In our case here, the idea that the solution will ultimately be delivered as a Web-centric application appears in the detailed requirements. For example, the statement in the main flow of the `Manage Time` use case (discussed later), "The employee should have access to a visual calendar to help select and compare selected dates," is prompted in part by the knowledge that most Web browsers do not have a common date picker or calendar widget. In the specification of a native Windows client application, this assumption of such a capability might not have been explicitly brought out since such controls are in common use on Windows-based native client applications. The use case writer in this case is explicitly identifying an area where the system needs to be user friendly. A more subtle and pervasive indication that knowledge of the architecture is known during the process of use case writing is the constant reference to "navigating to Web pages" and "submitting" information, concepts linked tightly with Web-centric architectures.

Sadly, there is no commonly agreed-upon way to quantitatively represent an arbitrary architecture. Here, we will use the 4+1 architecture view model [3], as described in Chapter 6.

The following brief descriptions of a Web application's architecture are not meant to be a complete discussion of Web-centric architectures; however, we do cover enough significant aspects here to help explain design decisions made later.

The Deployment View

A Web application, being a specialization of a client/server application, has minimally two main nodes, the server and the client browser. The server is a node that

has a known address on a network and is configured to listen for HTTP requests on a specific port, typically port 80. A client browser application makes a request, at the behest of the user, for an HTML-formatted resource on the server. The server, most likely, will be concurrently running a number of services, including other Web applications, possibly a database server, an application server, and so on. In Figure 12–2, the `Client` and `Server` nodes are clearly identified.

In the Deployment View of the key components, execution environments `Tomcat` and `Cloudscape` are treated as nested nodes of the server. The `Tomcat` node is a Web application execution environment based on the Java environment. The `Tomcat` node itself is shown here deploying the artifact `VTSWeb.war`, a Web application archive file. The `Cloudscape` execution environment is a database server capable of executing SQL files and shown here deploying an artifact called `VTS.sql`.

The `Client` node in Figure 12–2 is shown deploying the `Firefox.exe` artifact, which is an executable HTML browser application. The `Client` node has a communication link to the `Server` node. This communication link may be anything from a dialup ISP to broadband or wireless. The important and logical aspect to concentrate on is that the client communicates to the nested Web and database servers via this communication link.

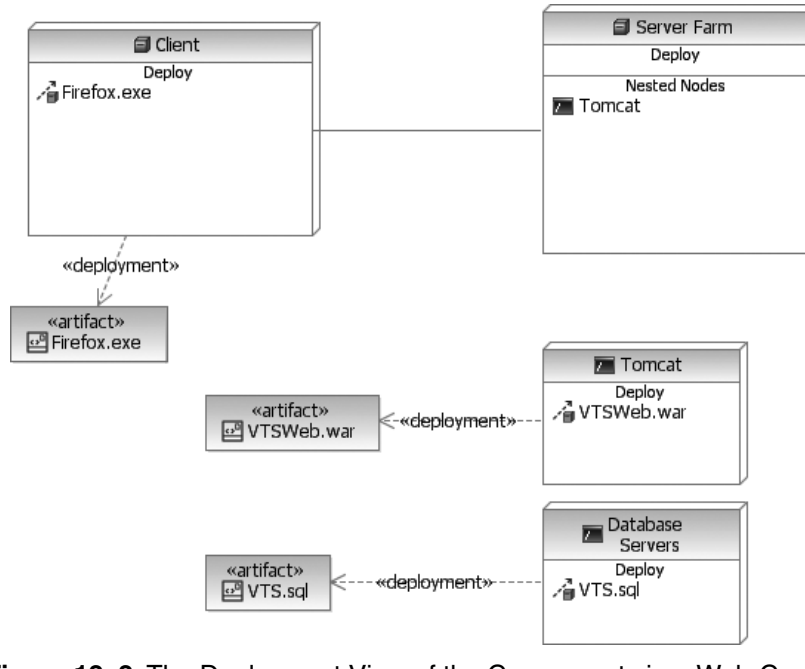

Figure 12–2 The Deployment View of the Components in a Web-Centric System

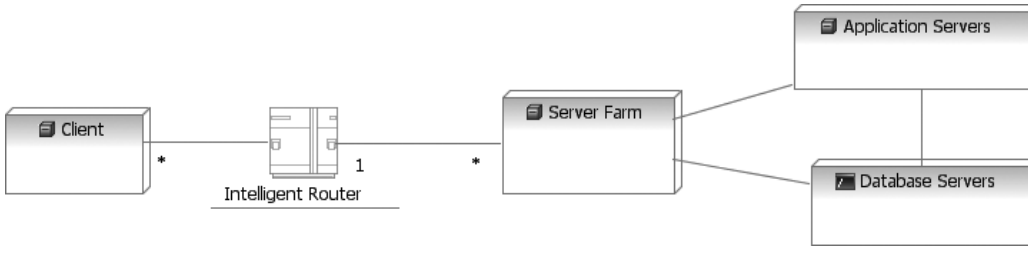

Figure 12–3 A Deployment Diagram for a Web Application Server Farm with Parallel and Redundant Processing Nodes

This representation of a Web-centric system is simplistic. In larger systems, the deployment diagrams showing the topography of all the nodes, components, servers, and communication links can be complex, resulting in the need for a good visualization and documentation system such as the UML. Figure 12–3 is an abstract deployment diagram showing how an intelligent router can be used with a Web server farm and how an entire set of applications can be run from application servers. The details of such strategies can be quite complex, but the general strategy of scaling an application by adding additional processing nodes is certainly viable in Web applications.

The Logical View

The typical Web application has at least four logical components: the browser running on the client, the Web or application container, a separate component for the application logic itself, and a database server component. In Figure 12–4, the `Firefox.exe` component is a commonly available multiplatform browser, and `Tomcat` is a popular Web container based on the Java Server Pages (JSP) specification. The `VTSWeb` component represents the business application, and `Cloudscape` is a simple portable database server. The most important logical point of this diagram is that the client browser is never in direct contact with the database or even the business application component. All access to server-side resources is mediated by the Web container. This makes the Web container an important component to consider when thinking about security requirements.

Web servers are designed to listen to certain ports, usually port 80, and respond to `GET` and `POST` requests. `GET` and `POST` are commands defined in the HTTP protocol. They are essentially two simple ways to request information from a Web server. The `POST` method is a little more extensible and used more often when data or files supplied by the user are sent to the server. The determination of which command to use is embedded in the HTML-formatted page being rendered by the browser. Usually the information returned by a browser request is an HTML-formatted document, which can be visually rendered in the browser, but

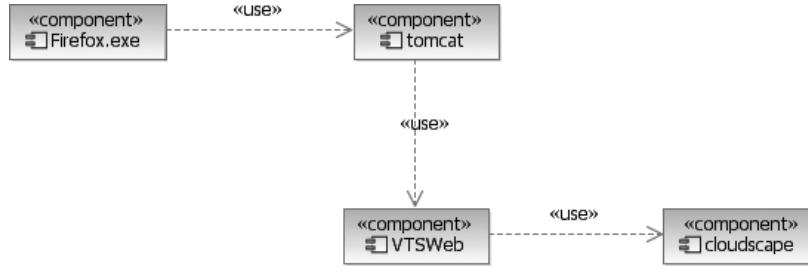

Figure 12–4 The Logical View of the Primary Components of a Web Application

the server could return streamed data in any form, leaving the client responsible to save or delegate the processing of the information to another locally installed application.

Figure 12–5 shows logical classes representing elements in both the client and server tiers of the application. On the client, a browser is responsible for requesting Web pages (or more generically, resources) with simple HTTP `GET` or `POST` commands. These services can be provided by the operating system or may be implemented in the browser itself (in which case the browser calls on lower-level network APIs). The Web container is constantly listening on certain ports (e.g., port 80) for incoming HTTP requests. An HTTP request is packaged by the browser and sent to the Web container. It contains a resource identifier that points to a specific Web page or references a Web application. The request can have accompanying it a set of key-value pairs (parameters), all represented as strings.[1] Some HTTP requests are packaged with more complex form data that the user supplies just before the page request is submitted.

When the Web container receives a request for a resource, it must first determine the file or application resource to invoke. The container invokes the proper resource, and if the resource indicates that it is dynamic and should be processed, the Web container executes it. During processing, the HTTP request information (parameters and form data) is examined, and the application performs the necessary business logic. Often this logic is executed in a separate application server (e.g., an EJB container), which potentially accesses another database server.

Logically, a Web application is made up of Web pages, controllers, and entities (the three basic stereotypes found in an analysis model). Static Web pages (no

1. The gory details of parameter and post data formatting can be found at the World Wide Web Consortium Web site (www.w3.org). Fortunately, most of these details are managed by the Web application framework on which we choose to implement the application.

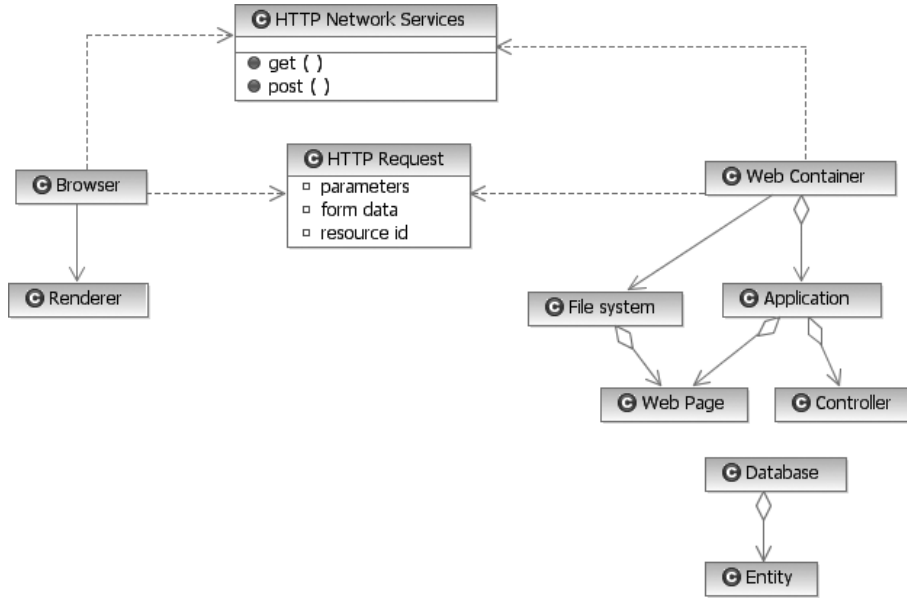

Figure 12–5 The High-Level Logical View of the Objects Involved in a Web Application

processing) may just reside in a file system, while Web pages that contain business logic processing must be loaded and executed in the context of a container. Controller objects are often embedded in components, and persistent entities are managed by databases.

The Process View

There are minimally two, but usually more, processes involved in a typical Web application. The client and server operate asynchronously, except during the handling of HTTP requests. Additional database, authentication, and messaging servers are often part of a typical Web application. They may coexist on a single server node or be distributed across multiple nodes. The process layout of a Web application has the same flexibility as any client/server architecture, with only the one requirement that client nodes must run some form of Web browser client software to initiate communication with the server-side application.

As intimated earlier, the most important thing to understand about Web application architectures is that they work in a connectionless mode. That is, the client and server are never connected longer than it takes to process the one `GET` or `POST` request. Once a server resource (i.e., HTML-formatted Web page) is requested by a browser and the server responds with that resource, the connection

Client State Management

One of the most interesting problems that Web applications face is that of managing client state on the server in a connectionless environment. Since every request and response made between client and server is completed with a new and unique connection, it is difficult for a server to keep track of the sequence of requests of any one particular client.

Managing state is important for many applications since a single use case scenario often involves navigating through a number of different Web pages. If there were no state management mechanism, you would have to continually supply all previous information entered for each new Web page. Even for the simplest applications this can get tedious. Imagine having to reenter the contents of your shopping cart from scratch every time you visit it, or entering your user name and password for every screen you visit while checking your Web-based e-mail.

The most commonly implemented solution to this problem, originally proposed by the World Wide Web Consortium (W3C), is the HTTP State Management Mechanism or, as it is more popularly known, cookies. A cookie is a piece of data that a Web server can ask a Web browser to hold on to and to return every time the browser makes a subsequent request for a HTTP resource from that server. Typically, the size of the data is small, between 100 and 1000 bytes. Web application frameworks manage client state by generating a unique identifier each time a browser first interacts with the application and places this ID in a cookie for the browser. This ID is used as a key into a map on the server that contains all the state information for that particular client.

URL redirection is an alternative approach to cookies for managing client state. In this approach, every hyperlink and form places the ID on the end of the URL submitted to the server during the next page request. The ID performs the same role as a cookie, but as a parameter of a URL it can be used with browsers that have explicitly turned off their cookie feature. The downside to this approach is that every page in a Web site must be dynamic, and the user cannot wander outside of the application in the middle of a use case scenario, lest the particular session between the client and server be broken.

The two other approaches for managing client state serialize the entire state into the URL parameters or into cookies. These are not commonly used for a number of reasons, the least of which is that they are inherently less secure since the client's state is sent over the wire back to the browser, whereas in the previous approaches the state is managed solely on the server. Also, when all state values are placed in cookies, they are limited by size (4K) and can have at most 20 cookies per domain. All state data must be encoded into simple text (no white space, semicolons, and so on). This also implies that you can't easily capture client state with higher-level objects in the session state.

between client and server is broken. Figure 12–6 shows a client making a simple `GET` request to the server (`Tomcat`). The server determines from the request which Web application and resource to invoke. A Web page inside the application is instantiated or invoked, and this represents the main trigger for the execution of business logic. Nearly all business logic in a Web application is invoked during the process of handling a `GET` or `POST` request. When the logic is finished, the application is responsible for preparing a response page (i.e., the next Web page in the scenario). It is important to realize that once the request for a Web resource is fulfilled, the application stops working for that particular client.

The implications are that it is not at all obvious how a server application can keep track of one particular client's request history and hence the relative internal state of the client. For example, without exploiting certain features of HTTP or implementing certain architectural mechanisms, a server application would have difficulty managing even a simple shopping cart or the state of a particular user walking through a multistepped wizard. Fortunately, most Web application environments provide many useful utilities and mechanisms for managing client state.

Another implication of this architecture is that the server does not know whether the user has abandoned the application in the middle of some business process. It is entirely likely that on occasion a remote user might become disconnected from the network and hence be unable to finish a particular business process that was started. In a more classic client/server application, the server might receive a notification that the client has been prematurely disconnected, but in a Web application, there are no notifications sent to the server when a user becomes disconnected or simply decides to just shut down the browser.

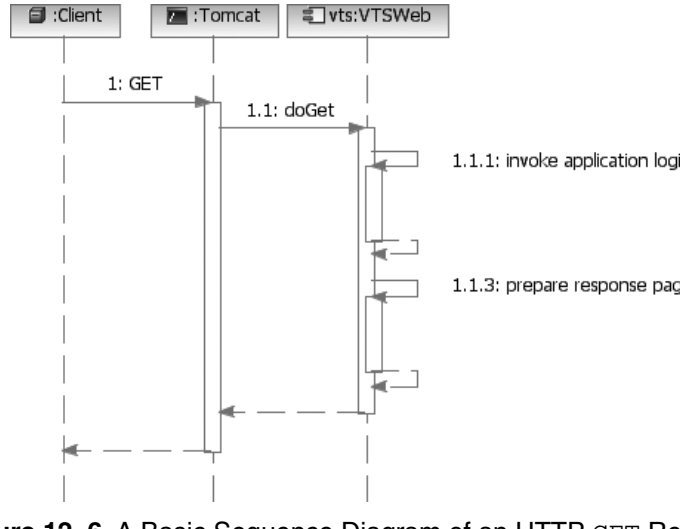

Figure 12–6 A Basic Sequence Diagram of an HTTP `GET` Request

Web application designs therefore must be very mindful of what resources are opened and accessed between Web page requests. For example, one cardinal rule of Web application design is to never open a transaction in one page and close it in another. The time between Web page requests from a single client is usually on the order of seconds and could at any time abruptly stop. Managing transactions and locks on this order of magnitude is surely going to bring an application server to its knees.

The Implementation View

Figure 12–7 shows an overview of the implementation model for the Vacation Tracking System and describes the system's tiers and packages. In this diagram, the main Web tier component, `VTSWeb.war`, is shown containing Web page artifacts (JSP and HTML files) as well as a set of Java classes in the `web` and `sdo` packages. This code is used to process and invoke the EJB logic in the business tier. The package `com.acme.vts` is shown, in the background, as the main namespace for all the Java components. The Deployment View would describe in more detail the deployment of the components in the tiers, and the Logical View would describe in more detail the nature and responsibility of the components.

The Use Case View

The Deployment, Logical, Process, and Implementation Views are tied together by implementing the basic stimulus/response use case. A client makes an HTTP

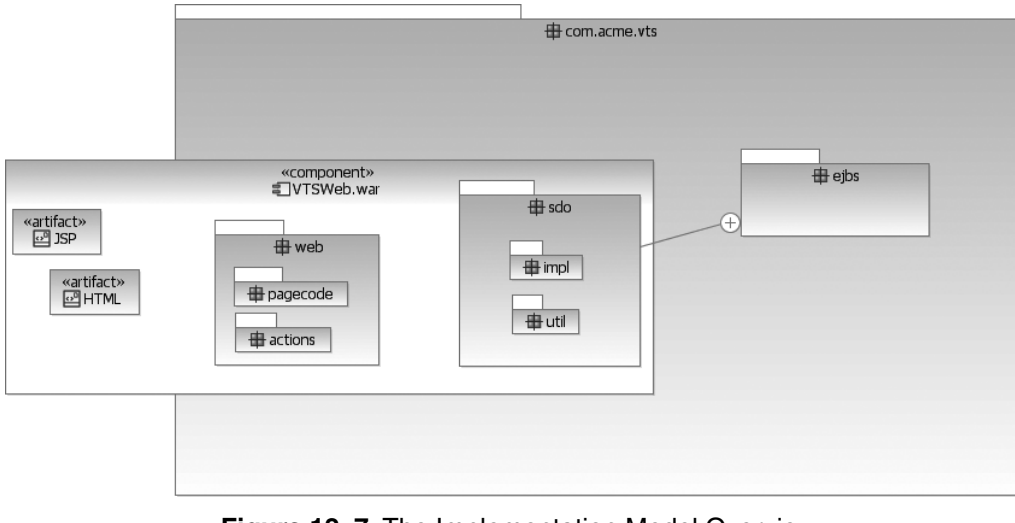

Figure 12–7 The Implementation Model Overview

request to a server for a Web page. The server examines the request and determines which application or resource needs to be loaded and executed. Some resources result in business logic processing, while others simply display static data. When the processing is complete, the application is responsible for composing a response, usually in the form of an HTML Web page that can be rendered in place of the previous Web page. This response page contains new information and options for the user to invoke or request. By assembling an entire collection of these Web pages, each specialized to display and accept information that is part of the application, an entire business process can be implemented.

In this chapter, we focus on one architecturally representative use case: `Manage Time`. This use case by far is the most frequently invoked and the one most viewed by all the actors of the system. As a result, it is critical to implement this use case effectively and to ensure that it meets all of the overall design goals, including the ease-of-use feature.

Many different templates can be used for use case specifications. The format used in this chapter shows more robust steps in the flows. This avoids the stimulus/response style flow that Web applications can create. Also, this style of use case specification is good for situations when there are complex or robust alternate flows. The style or format that you ultimately use is up to you and the level of formality to which your organization adheres.

The following example gives a summary of the written specification for the `Manage Time` use case.

Use case name: `Manage Time`

Actor: `Employee`

Goal: The employee wishes to submit a new request for vacation time.

Preconditions: The employee is authenticated by the portal framework and identified as an employee of the company with privileges to manage his or her own vacation time.

Main flow:

1. The employee begins by selecting a link from the intranet portal to the VTS.
2. The VTS uses the employee's credentials to look up the current status of all the employee's vacation time requests and outstanding balances. Information is displayed for the previous 6 months and up to 18 months in the future.
3. The employee wants to create a new request. The employee selects one of the categories of vacation time with a positive balance to use.
4. The VTS prompts the employee for the date(s) and time for which to request vacation time. The employee should have access to a visual calendar to help select and compare chosen dates.

5. The employee selects the desired dates and hours per date (e.g., four hours might indicate a half-day vacation time request). The employee enters a short title and description (no more than a paragraph in length) so that the manager will have more information with which to approve this request. When all the information is entered, the employee submits the request.
6. If the submitted information is incomplete or incorrect or does not pass validation, the Web page is redisplayed, with the errors highlighted and documented.
7. The employee has an opportunity to change the information or cancel the request.
8. If the information is complete and passes validation, the employee is returned to the main VTS home page. If the employee's vacation time requests require manager approval, an e-mail is immediately sent to the manager(s) authorized to approve the employee's requests.
9. The vacation time request is placed in a state of pending approval.
10. The manager responds to the e-mail by clicking on a link embedded in the e-mail or by explicitly logging into the intranet portal and navigating to the main VTS home page.
11. The manager may be required to supply necessary authentication credentials to gain access to the portal and VTS application.
12. The VTS home page lists the manager's own vacation time requests and outstanding balances but also has a separate section listing requests pending approval by subordinate employees. The manager selects each of these one at a time to individually approve or deny.
13. The VTS displays the details of the requested time and prompts the manager to approve or disapprove the request. If the request is disapproved, the manager is required to enter an explanation. Once the manager submits the result, the internal state of the request is changed to approved or rejected.
14. Whether a request is approved or rejected, an e-mail notification is immediately sent to the employee who made the request. The manager's screen returns to the main VTS home page, and the manager may approve other outstanding requests, make a request for him- or herself, or simply leave the VTS application.

Alternate flow: `Withdraw Request`

Goal: The employee wants to withdraw an outstanding request for vacation time.

Preconditions: An employee has made a vacation time request, and that request has yet to be approved or denied by an authorized manager. See also main flow preconditions.

1. The employee navigates to the VTS home page through the intranet portal application, which identifies and authenticates the employee with the privileges necessary for using the VTS.

2. The VTS home page contains a summary of vacation time requests, outstanding balances per category of time, and the current status of all active vacation time requests for the previous 6 months and up to 18 months in the future.
3. The employee selects a vacation time request to withdraw, one that is currently pending approval.
4. The VTS prompts the employee to confirm the request to withdraw the previously submitted vacation time request.
5. The employee confirms the desire to withdraw, and the request is removed from the manager's list of pending approvals.
6. The system sends a notification e-mail to the manager.
7. The system updates the request state to withdrawn.

Alternate flow: `Cancel Approved Request`

Goal: The employee wants to cancel an approved vacation time request.

Preconditions: The employee has a vacation time request that has been approved and is scheduled for some time in the future or the recent past (previous 5 business days). See also main flow preconditions.

1. The employee navigates to the VTS home page through the intranet portal application, which identifies and authenticates the employee with the privileges necessary for using the VTS.
2. The VTS home page contains a summary of vacation time requests, outstanding balance per category of time, and the current status of all active vacation time requests for the previous 6 months and up to 18 months in the future.
3. The employee selects a vacation time request to cancel, one that is in the future (or recent past) and has been approved.
4. If the request is in the future, the employee is prompted to confirm the cancellation. If the request is in the recent past, the employee is prompted to confirm the cancellation and provide a short explanation. If the employee approves the cancellation and provides the required information, an e-mail notification is sent to the manager, and the state of the request is changed to canceled. The time allowances used to make the request are returned to the employee. The employee can also abort the cancellation, effecting no changes to the current requests.
5. The employee is returned to the main VTS home page. The summaries are updated to reflect any changes made to the employee's outstanding vacation time requests.

Alternate flow: `Edit Pending Request`

Goal: The employee wants to edit the description or title of a pending request.

Preconditions: An employee has made a vacation time request, and that request has yet to be approved or denied by an authorized manager. See also main flow preconditions.

1. The employee navigates to the VTS home page through the intranet portal application, which identifies and authenticates the employee with the privileges necessary for using the VTS.
2. The VTS home page contains a summary of vacation time requests, outstanding balances per category of time, and the current status of all active vacation time requests for the previous 6 months and up to 18 months in the future.
3. The employee selects a request to edit, one that is pending approval.
4. The VTS displays an editable view of the request. The employee is allowed to change the title, comments, or dates. The employee can also choose to delete or withdraw this request.
5. The employee changes request information and submits the changes to the system.
6. If the employee withdraws the request, the VTS prompts for confirmation before withdrawing the request. If changes are made only to the information, the changes are accepted, and the screen returns to the main VTS home page. If there are errors or problems with the information changes, the VTS redisplays the editing page and highlights and explains all problems.

This use case description contains quite a bit of information about the proposed VTS and how it is expected to be used by a typical employee. It is for the most part a functional description of the system from the viewpoint of the `Employee` actor. Indeed, this is exactly what a use case is supposed to be. Unfortunately, this description is insufficient to even begin analysis with. What is needed are the nonfunctional requirements and much more information about the domain. Nonfunctional requirements typically include requirements on the environment, performance, scalability, security, and so on. Domain knowledge can be in the form of discrete requirements, for example:

- All employees work eight-hour days.
- Each employee's vacation time requests are subject to the restrictions of each employee's primary work location in addition to overall company policies and restrictions.
- Vacation time request validation rules are defined and owned by the HR department.

These types of requirements or knowledge may be found embedded in use case specifications, or they can be captured as discrete line item requirements. This type of domain knowledge may also be referenced by commonly accessible

documents. For example, the detailed policies and rules for validating a vacation time request are part of a company's employee manual and most likely available through a variety of sources (e.g., intranet, forms, documents, new employee orientation presentations, and so on). A project's requirements set would simply reference these existing documents rather than try to duplicate them.

12.3 Construction

The most important task when analyzing a potential Web application, as with most types of software applications, is the identification of the system's entities and processes. The entities and processes of an application represent concepts in the business domain and are ideally independent of an architecture, but not necessarily so. In a Web application, one critical set of artifacts is the navigation map and Web page definitions. Roughly speaking, these elements correspond to the classic analysis stereotypes of entity, controller, and boundary. We'll begin with a Web-centric model: the User Experience (UX) model.

The User Experience Model

The UX model [1, 2] is one example of capturing the user interface elements of a Web application at a sufficient level of abstraction so as to express a concrete navigational map between the Web pages in the system, while ignoring the styles (fonts, sizes, colors, and so on) and other user interface specifics that are best developed later in the process. Figure 12–8 shows a high-level fragment of a UX model that describes the screens used to implement our primary use case. In this model and diagram there are two key stereotypes, «Screen» and «Form». A «Screen» stereotyped class represents a complete unit of user interface displayed to an end user or, roughly speaking, a Web page. Some screens contain HTML form elements, which are used to submit user-entered information back to the server. These stereotyped classes are always contained by a screen and when submitted result in navigation to another screen in the system. Directed association relationships indicate navigational pathways through the screens.

While Figure 12–8 is most useful for understanding the navigational flow through the screens of the system, it provides little in the way of screen content. More detailed diagrams that contain screen content are also useful at this level of abstraction. Figure 12–9 shows the content of the VTSHome screen. Attributes of a «Screen» stereotyped class indicate discrete data values, typically strings or simple types easily rendered in HTML. The employee name and current date attributes are probably used in a header or at the top of the screen. The message attribute is used to display an informative or error message after an

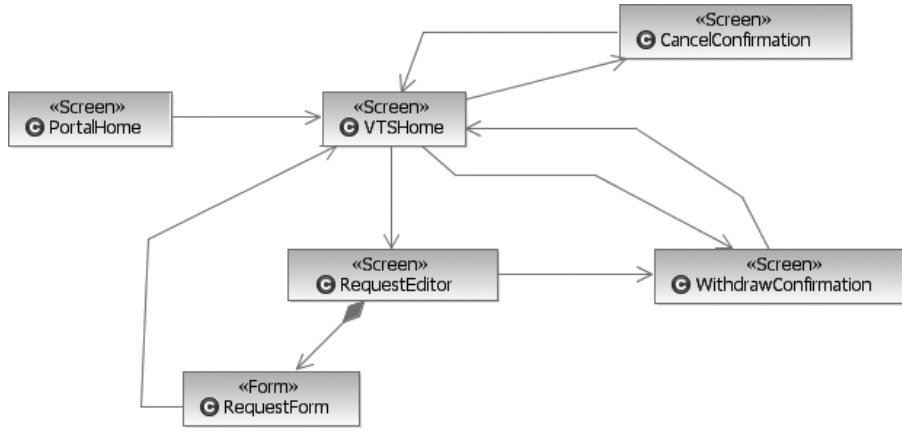

Figure 12–8 A High-Level User Experience Model

action has taken place. For example, when the employee completes a new vacation time request and returns to the home screen, the message might say something like "Your request requires the approval of your manager, who has been notified by e-mail."

Often content in a screen is multivalued and complex. To accurately represent this type of content, we define another class stereotype, `«Content»`, that when applied to a class identifies a coherent bundle of information. Content items are often used as line items in a list. In Figure 12–9, the classes `Request` and `PendingApproval` are modeled as content items contained by the home screen. The screen potentially displays multiple instances of each. Content

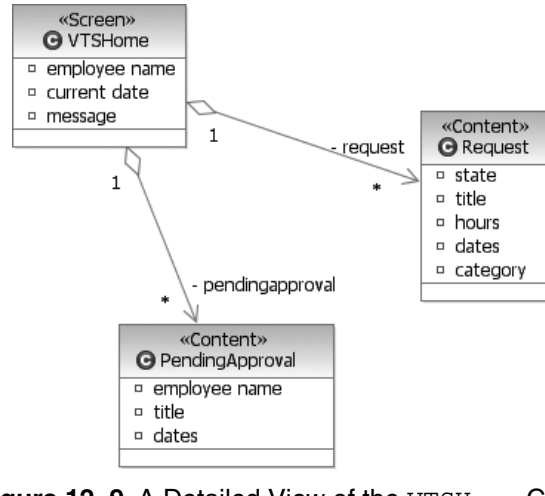

Figure 12–9 A Detailed View of the `VTSHome` Class

classes define attributes that are like the attributes of a screen, things that can be easily rendered in HTML. It is not important at this high level of the model to worry about actual data types. What is important is to simply define them with a name and short description.

It is interesting to note that even at this early point in user experience design, some decisions are being made that will have a significant impact on the final design. By specifying the content of the `VTSHome` page not only to contain the summary of current vacation requests for the employee but also to have this same page used by the manager to view pending approvals, we are implying that this screen is the home page for both employees and managers. Nowhere in the use case specifications nor the nonfunctional requirements was it stated that employees and managers should use the same home screen. Such an assumption is not without reason since managers are also employees and can do all the things (with respect to the VTS) that employees can. Decisions like this are exactly what the UX model is intended to document. Ultimately, the final design will determine the implementation; however, from the logical (and user experience) point of view, employees and managers share the same home screen.

The Analysis and Design Models

In analysis, we want to capture the entities and processes of the system. Since we know that this system will be implemented as a Web application, we are less concerned with boundaries because they are explored in the UX model. The analysis model is a first attempt at identifying the elements that will make up the solution space but using the vocabulary of the domain. This means that most of the elements of the analysis model will have names that correspond to things and activities that are described in the requirements and are recognizable by the domain and end users.

One easy way to get started is to do a simple noun-verb analysis of the use case specifications and related requirements documents. Important nouns tend to represent classes in the model, while action phrases (verbs) tend to be represented by operations on those classes. Attributes and relationships represent natural intrinsic properties of the classes captured in the model. Figure 12–10 represents an initial start, focusing on the main domain classes and concepts represented in our primary use case.

There are several important points to consider in this class diagram. First is the idea that vacation time requests are separate and distinct from vacation time grants. A grant in this context represents available time for the employee to draw on when requesting time off. Grants are administered by the HR department and determined by company policy. In the context of this use case, however, their pri-

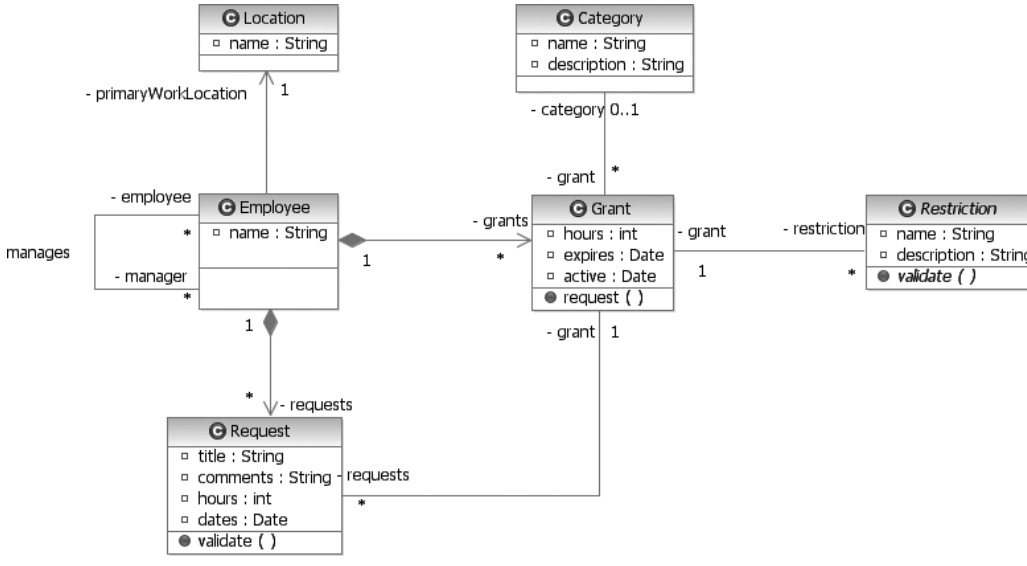

Figure 12–10 An Analysis Model Class Diagram Supporting the Primary Use Case

mary responsibility is to create new vacation time requests. Their persistent properties are the number of hours that can be requested per calendar year and when the grant expires (if at all).

Another important decision captured in this diagram is the idea that a manager *is an* employee. While it may sound obvious to everyone familiar with the popular usage of these common business terms, such assumptions cannot always be made when developing software. The class diagram goes further to say that an employee can have multiple managers. This is not meant to be a statement on the company hierarchy and organization, but rather in this context it means that an approval of an employee's request for vacation time may come from multiple sources. For example, in some companies, high-ranking executives have dedicated personal assistants. A personal assistant may be delegated to make vacation time decisions for the executive.

Other inherent properties of an employee are the name and primary work location. Note that at this level of abstraction and required usage, an employee name need only be managed with a single string data type. Other applications that manage employee information will most likely require separate values for the title and the first, last, and middle names, but in this application those distinctions are not required since the only place the employee name is used is in the display of the home screen. As a general rule, unless the problem absolutely requires a complex solution, don't propose one.

Another interesting and important aspect of most analysis models is the lack of identifier definitions. In nearly every implementation of a system with persistent entities, the concept of a unique ID is present. For example, most companies assign an employee ID to each employee. This ID uniquely defines that particular employee and is often reused if the employee leaves and then returns to the company. The need for the ID is clear given that there is no guarantee of the uniqueness of names. Yet in our model, there are no ID attributes defined. That is because in most analysis models, it can be safely assumed that IDs and other properties necessary to implement the analysis with the architecture will be added as required. Unless there is a very specific business need for information like this, there is no need to specify it during analysis. For example, suppose we had in our proposed system another actor called `Auditor`, who was responsible for browsing all the vacation time requests and checking for regulatory compliance; that actor would in fact need and expect employee IDs to cross-reference with other employee data. In this case, it would be important to explicitly specify this attribute of an employee.

Some interesting elements in the model can be elaborated with a state machine. For example, the `Request` class has a defined state machine, as shown in Figure 12–11. As far as state machines go, it is not that interesting; however, it is significant. It tells us that state is important for a `Request` and that it is well defined. In the diagram shown in Figure 12–11, we see three transitional states and four final states defined. The paths through these states are very simple and in this diagram are unlabeled.

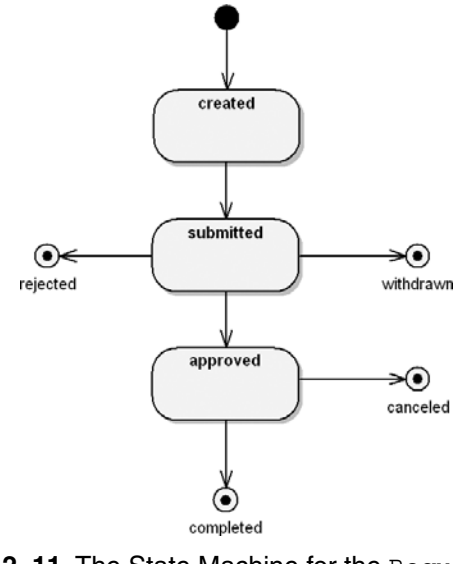

Figure 12–11 The State Machine for the `Request` Class

By far the most difficult part of the analysis was creating the elements related to ensuring that any requested vacation time passed all the rules and restrictions of the company and primary work location. Not much is stated in the use case specifications about these rules; instead, the requirements simply reference internal company documents as current examples. The requirements do state that there is to be a rules-based system managed by the HR department. This statement alone indicates that the approach must be flexible and manageable by users whose skill sets are not in computers or algorithms. This is important because one obvious and very flexible mechanism for implementing a general-purpose rules-based system is to enable the capture and interpretation of a flexible scripting language like JavaScript as a rule. Most algorithms, complex or simple, could be implemented with JavaScript and easily interact with persistent entities and objects. The principal problem with this is, of course, that the typical HR clerk, being charged with the responsibility of maintaining these aspects of the system, does not typically have JavaScript writing as a primary skill.

Long before we choose an implementation solution or even a strategy, we need to first identify the higher-level abstract elements that belong in the model. For now we can create a single class called `Restriction` and place on it all the responsibility for validating a vacation time request against the company and location-specific policies. A deeper understanding of the rules and regulations for vacation time is needed. Unfortunately, not much about the inherent structure of a rule is documented in the use case specifications, and only specific instances of the rules are captured in the company documents. Therefore, analysis leads us to study in detail and look for patterns in the current set of rules. Analysis activities like this often require interaction with domain experts.

After a careful examination of the rules implemented in company policy, and after speaking with the domain experts in the HR department, we can assemble a summary of the major rule types.

- An employee can't take more than *X* consecutive days of leave for *Y* type of grant.
- Vacation time of type *X* cannot be taken when directly adjacent to a company or location-specific holiday.
- Vacation time of type *X* is limited to *Y* hours per week or month.
- Vacation time may not be granted when there are only *X* number of employees scheduled to work from the list *Y* of employees.
- Vacation time may not be granted on these dates: *X*.
- Vacation time of this type is limited to certain days of the week: {M, T, W, Th, F, Sat, Sun}.

This more detailed understanding of potential rules for vacation time request approval will lead to some important changes to our model. It is clear that

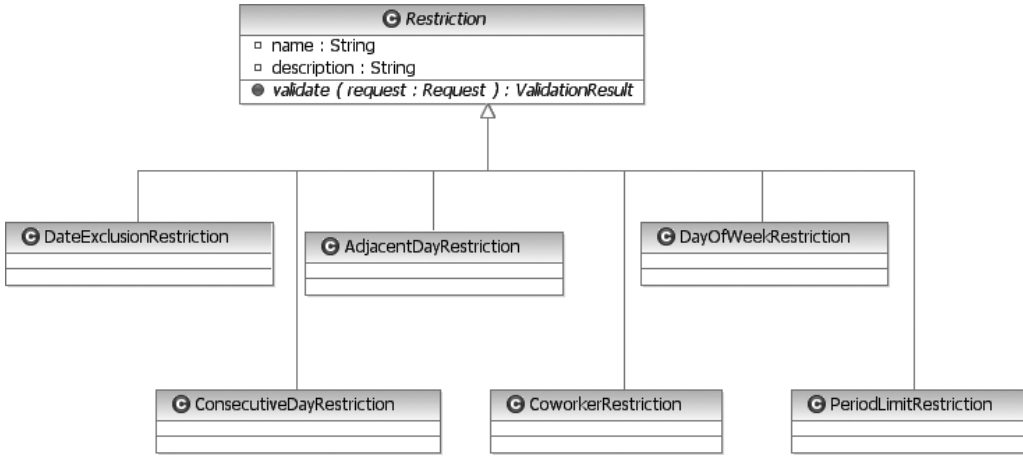

Figure 12–12 The `Restriction` Class Hierarchy

implementing restrictions is varied and requires specific, individually set information. In our attempt to formulate a solution strategy, we will create specializations of the now abstract `Restriction` class, one for each type of rule in the system, as shown in Figure 12–12.

Each specialization is required to implement the `validate()`, method which accepts a `Request` object as a parameter. From the `Request` object a `validate` algorithm can navigate to most of the information it needs (`Employee`, `Grant`, `Location`) to validate the request. Given the varied nature of restriction types, not all of the existing information is sufficient to make a validation. Therefore, each specialization will manage a set of properties or relationships to objects that cannot be derived or navigated to via the `Grant` object. For example, the `CoworkerRestriction` class will need to manage the list of coworkers as well as identify the minimum number of employees allowed to be scheduled for work (Figure 12–13). Rules like this are often associated with safety issues (e.g., there must always be at least one employee on duty who is trained in first aid).

The abstract `validate()` method initially returned just a simple Boolean indicating a pass or fail evaluation of the request. But if we look at the requirements a little closer, we see that the user needs to be notified with an explanation if he or she tries to submit an invalid vacation time request. Since the rules for validation

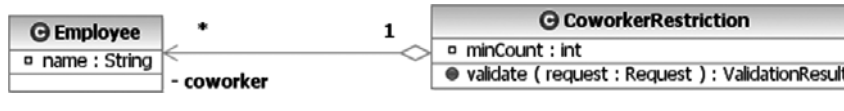

Figure 12–13 The `CoworkerRestriction` Class and Properties

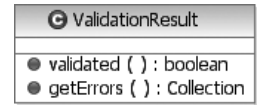

Figure 12–14 The `ValidationResult` Class

are now encapsulated in each of the `Restriction` specializations, it seems appropriate that these classes provide a mechanism to return this information to any calling process. Also, since more than one violation can occur, the results of a validation should be accessed as a collection. The result of these requirements leads us to the creation of a new class, `ValidationResult` (Figure 12–14), which is returned by the `validate()` method.

The `ValidationResult` class is actually quite simple. It defines two methods: `validated()`, which returns a simple Boolean indicating a validated request, and `getErrors()`, a method that returns a collection of string messages, each corresponding to a detected problem with the request. At the analysis level, it is sufficient to keep the class this simple. Let the activities design resolve the details of implementation. The important point here is that it must be possible for a process to validate a vacation time request, make each error result available, and provide a simple means of determining validity.

Another observation of the rules system that is evolving here is that restrictions may not be associated with a particular `Grant` or `Employee` object, but rather, are broadly applied to a `Location` or `Category` of `grants`. Thus our earlier analysis model must change (Figure 12–15). Establishing these relationships makes it easier for the HR department to apply a broad class of common rules without having to duplicate them for each employee. While it is clear this information can be navigated to during validation, the motivation for design change is

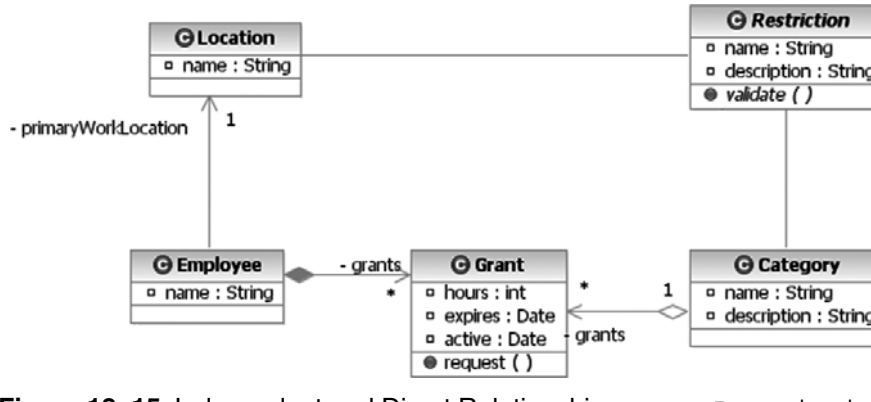

Figure 12–15 Independent and Direct Relationships among `Restriction`, `Location`, `Grant`, and `Category`

based on the overriding desire to make the system easier to use, and in this case the ease of use will be improved significantly for the HR personnel.

Now that the restrictions and rules are better defined, it may be time to revisit some of the HR use cases because now we know more information about the type and nature of information that will need to be managed by the HR department with respect to an employee's vacation time. Although beyond the scope of this simple chapter, a set of Web page screens that prompt for the unique and specific values required by each concrete type of restriction could be envisioned. In such a mechanism, it would be necessary for `Restriction` objects to publish their required parameters (i.e., the *X* and *Y* placeholders in our rules summary) so that the HR clerk could provide the values in a user interface. A helper class, `RestrictionParameterDescriptor`, is defined to encapsulate each of these values. Concrete implementations of `Restriction` are responsible for providing an array of these parameter descriptor objects. The descriptors could be used to prompt the HR clerk when defining new restrictions for an employee, location, or category. Finally, the abstract `Restriction` class should be able to accept and provide individual parameter values. This is accomplished through simple `put` and `get` parameter methods, keyed off of the parameter name. Figure 12–16 illustrates both classes.

At various points during analysis, it is useful to test the ideas being put forth in the model. However, since we are still independent of an actual concrete architecture for which we can write code, we must analyze our model a little more abstractly. We do this by executing thought experiments on the elements of our model, specifically, trying to understand in moderate detail how specific scenarios will play out given the structures and behaviors documented in the model. The area of our model dealing with restrictions is a good example of the need to verify that at least the typical and most important use cases can be implemented this way with realistic data.

A useful pair of tools that the UML provides us is object diagrams and communication diagrams. Our task here is to evaluate the ability of our model to validate a new vacation time request. We want to ensure that the behavior can be accomplished according to the requirements and with the set of defined classes and

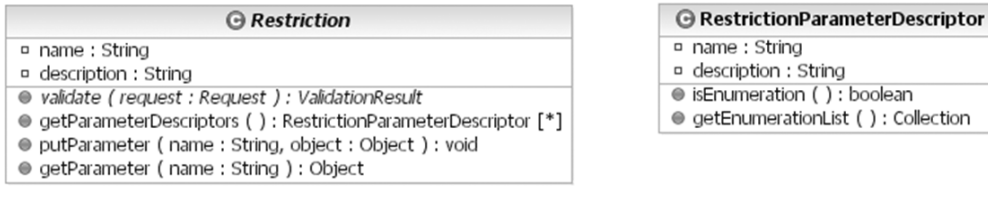

Figure 12–16 Making Restrictions Easier to Manage Programmatically and through a User Interface

objects of our model. In this thought experiment, we use these diagrams to understand how a new request for an employee named Jim who works in the North Factory might be executed.

Figure 12–17 shows the object diagram of a set of objects for our hypothetical employee, Jim, who wants to use a vacation time grant given as a bonus for completing his work on time. The figure indicates with dependencies other object instances that can be accessed or navigated to. The instances have real-life names, appropriate for the scenario, and are followed by a colon and the type or class of object they are. This diagram has been annotated with a few freeform rectangles to help emphasize and make clear the three sources of restrictions: `Grant`, `Location`, and `Category`.

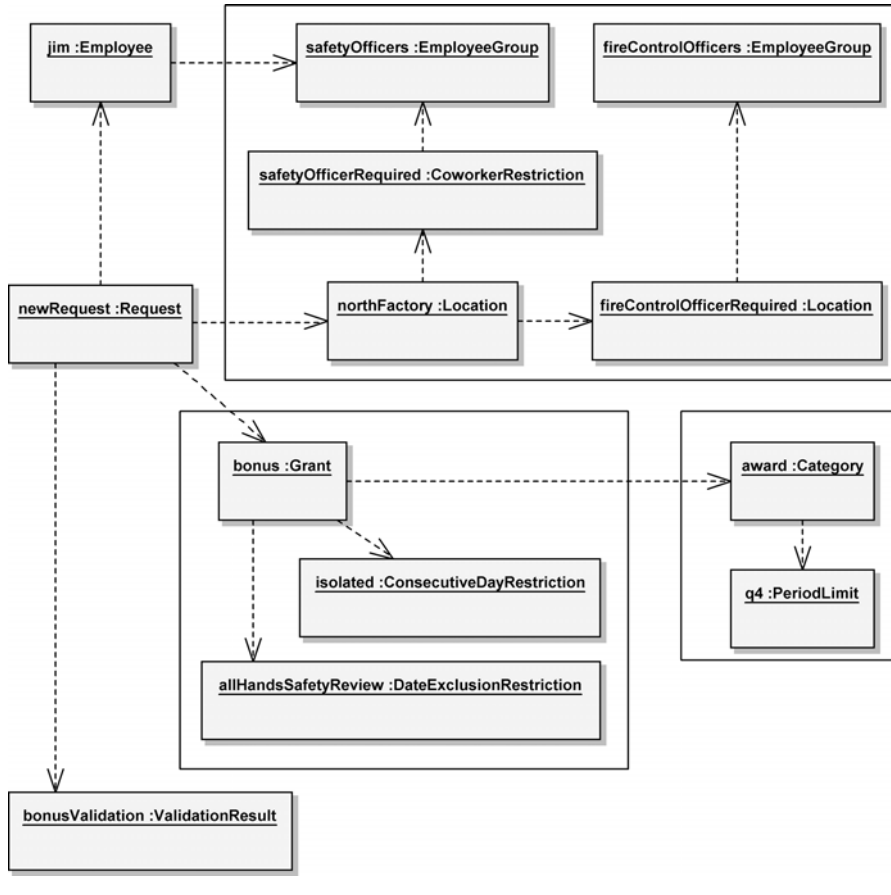

Figure 12–17 An Object Diagram Describing a Request Validation Collaboration

In Figure 12–18, we use a communication diagram to examine the communication paths for a `Request` validation. The lines represent communication paths between object instances, not structural relationships like those found in class diagrams. We examine the behavior of this collaboration by looking at the numbered messages that travel over the communication paths. The main coordinator of the behavior is the `Request` instance, which does most of the work in this scenario. The general flow is to get and validate the various restrictions for the `Location`, `Category`, and so forth by calling the `validate()` operation on each of the restrictions. If a restriction fails validation, its message is appended to the validation result instance.

One interesting flow to follow is that of the `CoworkerRestriction`. In order for a `CoworkerRestriction` to validate, it needs to know not only which `EmployeeGroup` it represents but also which `Employee` to check compliance for. A quick look at the `validate()` signature reveals that the `Request` object is passed in as parameter, and from it we can navigate to the `Employee`. However, these restrictions also need access to the vacation plans of all the other

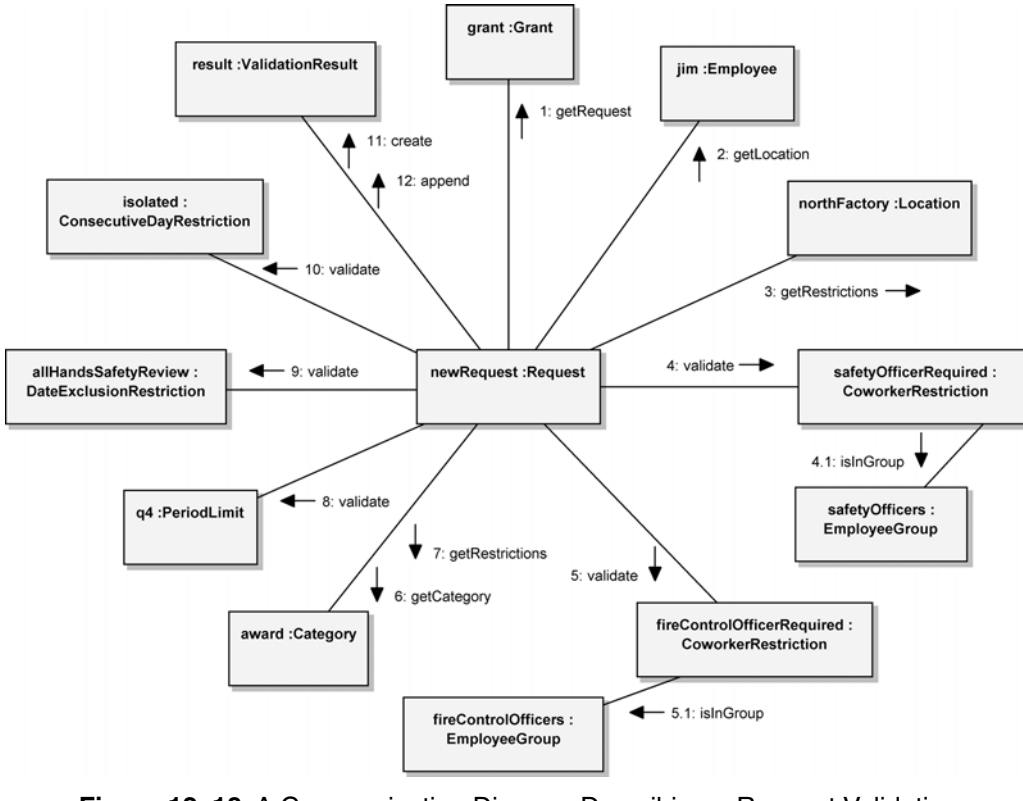

Figure 12–18 A Communication Diagram Describing a Request Validation Collaboration

employees in the group to ensure that at least one employee of the group is scheduled for the location at the request time. So far, our model has not adequately expressed how this can be accomplished. Therefore, as a result of this thought experiment, we need to revisit the model and make sure that these types of restrictions can get the information they need to validate a request.

As discussed in Chapter 6, the beginning and ending of analysis is not a milestone event. Analysis activities often take place throughout the entire development lifecycle, but for the most part they are concentrated at that time when most of the requirements are at least understood. When the analysis model has sufficiently matured, it is time to begin design. This is clearly a vague determination, but that is the reality for most project efforts. The decision to start design activities is often made on criteria that are very local, for example, the history of the project team and organization or the maturity of the architecture.

Now we convert the analysis into a design that can eventually be executed as software. Some design models are identical to executable models (or implementation models), while others are sufficiently abstract as to still require a certain amount of skill and effort during implementation. Web applications in general have a mix of these. Most business tier elements (entities and controllers) map quite closely to the resulting code in the system. Most presentation models, however, require significant effort during implementation before they are ready for the final system.

In this chapter's discussion of the VTS's design, we don't have the space to cover even a brief overview of all the interesting aspects and issues of design in Web-centric applications. Instead, we will discuss just a few of the most important design points, in particular those that are most unique to Web application architectures.

In addition to the basic architectural components of the Web, this project's architect has decided to use the following components and technologies:

- *Client tier*: Any HTML 3.2 capable browser
- *Presentation tier*: Java Server Pages (JSP), Servlets, and Java Server Faces (JSF)
- *Business tier*: Enterprise Java Beans (EJB) 2.x (specifically, Container Managed Persistence [CMP] beans for entities, and session beans for controllers), and Service Data Objects (SDO)
- *Security*: Central Authentication Service (CAS)[2]

The decision was made to use Java-based products in the overall architecture. This is not because Java-based solutions match up with the stated requirements

2. For more information, see www.ja-sig.org/products/cas/.

better than any other existing technology, but rather because they happen to match up better with the organization's current skill sets and tooling, in addition to being well suited to the task. This organization has built several Java-based enterprise applications with the core J2EE technologies successfully in the past. These technologies are familiar, and the tooling required to build and test these types of applications already exists on the developer's workstations. In our situation, there is no compelling reason why the architect should switch entire technology frameworks. There are only two reasons why an architect should even consider such a significant change.

1. Recent experiences demonstrated serious problems with the technology or an inability to meet the needs of the development team, maintenance team, and end users.
2. The technology itself is no longer being supported or is evolving so significantly as to make it appear like a new technology.

In the client and presentation tiers, the primary design goals are to implement an intuitive user interface that has a quick response and is easy to navigate. The interface should not have any dependencies on specific browser versions or features, or to put it more accurately, this application should behave properly on all standard HTML 3.2–capable browsers. Even though the current version of HTML at the time of this writing is version 4.01, the decision to support version 3.2 (and later) is driven not so much by the availability or popularity of browsers but by minimum required functionality. It has been determined that all of the features necessary to implement this application can be accomplished with the older version of the HTML specification. Therefore, since there is no real need to require new browsers and exclude the older versions, the application can support a broader selection of client configurations.

In addition to the core presentation technologies, JSP and Servlets, the architect has decided to use JSF components to assist in the development of the user interface. JSF is an API and custom tag library that contains a number of components for displaying forms in HTML interfaces, accepting and validating input, and assisting in page navigation. An interesting point to note here is a decision not to use the Apache Struts framework. Struts is an older model-view-controller (MVC) framework that is still very popular in JSP applications. Struts and JSF have been shown to work very well together, even if some of their functionality overlaps. While Struts does provide some of the functionality of JSF, its main strong point is its controller features. Struts is especially useful when a Web application implements long (i.e., many Web pages) business processes. It has been determined that our application's business processes are relatively short, and the value that the Struts framework would bring is not significant enough to warrant the extra layer of complexity required to implement it.

The business tier, where most if not all of the application's logic is executed, will run inside a J2EE Web and EJB container. The decision of whether or not the Web and EJB container will run on the same machine, whether they are clustered, or other such configuration information, is not a consideration of the design, except for the fact that such decisions can be made. One advantage of choosing a J2EE-based architecture is the ability to make such deployment choices late, as the system is initially deployed, or to effect these changes as a result of changing demand. The important thing to remember during the design and implementation activities is to not create or code anything that would prevent the normal ability of the J2EE container to shift or deploy components. Such development practices are often implemented in coding or design guidelines that the development team must follow.

The decision was made to use the CMP mechanism for storing all persistent data. There are many reasons to use this mechanism and many reasons why someone might not want to. Other options for persistence include the use of Plain Old Java Objects (POJOs) and a relational object persistence layer like Hibernate, Castor, or the Apache ObJectRelationalBridge (OJB). These options tend to be lighter in weight and more efficient, especially when the application is running on a single node. The decision was made to use the built-in CMP mechanism of the existing container for persistence since the performance needs of this application are not significant enough to warrant another technology or extra layer. Additionally, the use of SDO for managing the flow of data between the presentation and business tiers was adopted to make it easier to manage entity data.

One piece of this application's architecture that is not easily discarded is the use of the CAS for implementing single-sign-on. The idea of single-sign-on is that, while in the same browser, an end user need provide authentication credentials only once to access a whole range of related or unrelated Web applications. It was determined early while creating the vision that the VTS was to be an extension of the existing intranet portal system for the company, and that system currently uses CAS for authentication management. Therefore, the new VTS application will also have to use CAS to identify and authenticate end users.

Entities

The most important part of designing and implementing entities is identifying them. In any given analysis model, any objects with attributes, and even some without any defined attributes, may be designed as entities. The trick is to choose only those classes in the model that really need to be designed as persistent entities. For example, some analysis classes, even those with defined attributes, may merely be transient classes or value classes. For example, the `ValidationResult` class is used only as a return value to the `validate` method. Validations are not

logged or recorded in this system, so the `ValidationResult` class does not need to be persistent. When a request is validated, an instance of this class is returned for the caller to examine the results of the validation. When finished, the results are no longer needed or referenced. Therefore, the `ValidationResult` class can remain a simple POJO.

The VTS will use CMP beans to manage all of its persistent entities. CMP has come a long way since it was first introduced. Initially there were a number of problems, specifically with relationships and in performance. However, today's newer specification and improved EJB containers have addressed many of these issues, making CMP a preferred mechanism for managing entities.

Recently, several design patterns have developed around CMP beans. It is generally recommended to use CMP beans to define only local interfaces. A local interface can be accessed only by another bean in the same container. Marshalling data across locally connected beans is much more efficient than doing so over an infrastructure that can support cross-node communication. These objects are instead accessed by a façade object that has local connections to the CMP but also publishes remote interfaces that can be accessed by other session beans and remote clients.

SDO is used to marshal data into and out of the presentation tier. These objects are created and managed by the façade objects in the business tier. SDO is the result of a recent collaboration between BEA and IBM, two big J2EE container vendors, and represents a general agreement to certain best practices when handling data in EJB applications.

Another important design pattern for CMP beans is that they should be designed with minimal, if any, business-level behavior. In fact, it is easiest to consider a CMP definition as little more than a thin wrapping around a database table. All business-level behavior is instead placed in session bean façade objects, which are responsible for orchestrating the state in the various CMP beans to accomplish a unit of business behavior. This has a direct effect on how analysis entities are designed and implemented.

We can begin understanding how all these design-level patterns and conventions get realized in code by looking at the `Category` class. This class is relatively simple because it defines only two persistent properties plus a relationship to instances of the `Grant` class. Fortunately, `Category` does not define any significant behavior at the analysis level. If it did, we would need to remove it from the entity and place it in one of the façade objects. What we are starting with here is a very simple entity.

Design entities are represented by `«Entity Bean»` stereotyped classes. These classes define their persistent attributes; primary key attributes are further stereo-

typed. The methods of an «Entity Bean» class are segregated into local and remote (although in this case Category does not define any remote methods, according to our design practices), as well as instance and class level. Each of these corresponds to the actual interface used to define them. In the Category class, we need two interfaces, CategoryLocal and CategoryLocalHome, to define the local methods that can be invoked on instances of the bean and on the home or factory object responsible for creating or finding instances. Figure 12–19 shows a UML-like diagram[3] of the Category entity bean with a relationship to the Grant entity.[4]

Working with a single model element for each entity makes it easier to understand their relationships; however, in the implementation, any given entity is in fact implemented with a number of separate and distinct classes, interfaces, or configuration files. Our Category class, for example, requires two separate interfaces (CategoryLocal and CategoryLocalHome) and one implementation class (CategoryBean), is packaged with other beans in a Java Archive (JAR) file, contributes to the EJB deployment descriptor file, and results in the definition of a table in the database (Figure 12–20). The important thing to realize is that there is typically a large fan-out of model elements as you migrate from analysis to design and implementation. One analysis element will often map to many elements in the implementation, all requiring coordination. Listing 12–1 shows a fragment

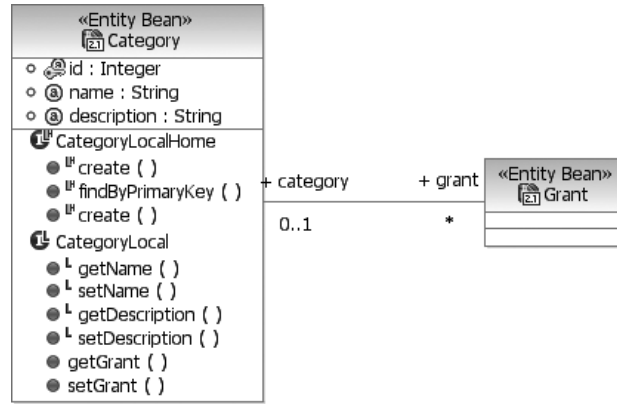

Figure 12–19 A Design Model Representation of the Category Class

3. This diagram is a customized view of an entity bean, which in a J2EE system includes the implementation class as well as its interfaces and some configuration information. Also, the use of getters and setters (methods that provide access to object properties) is not an object-oriented convention but rather one dictated by the use of Java and the J2EE framework.

4. Since this is a design model targeting a J2EE architecture, it is not inappropriate to use platform-specific notations.

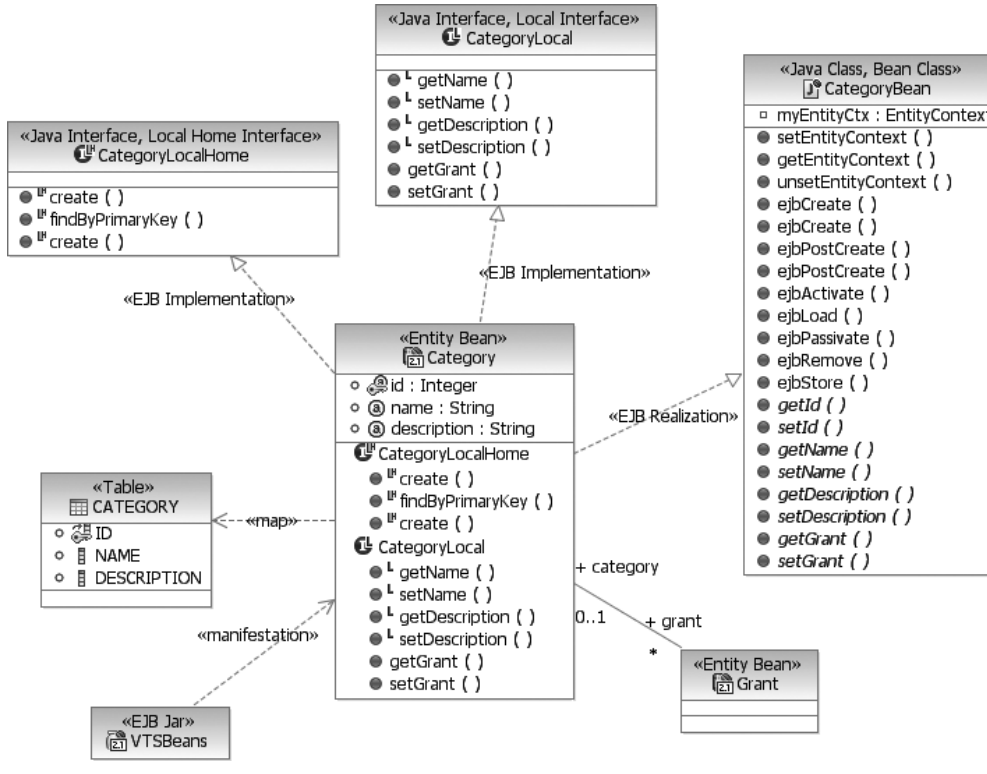

Figure 12–20 Design Model Elements for the `Category` Class

of the EJB deployment descriptor containing elements related to the `Category` entity.

Listing 12–1 A Fragment of the EJB Deployment Descriptor Containing Elements Related to the *`Category`* Entity

```
<ejb-jar id="ejb-jar_ID" version="2.1"
xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/ejb-jar_2_1.xsd">
  <display-name>VTSEJB</display-name>
  ...
    <entity id="Category">
      <ejb-name>Category</ejb-name>
      <local-home>com.acme.vts.CategoryLocalHome</local-home>
      <local>com.acme.vts.CategoryLocal</local>
      <ejb-class>com.acme.vts.CategoryBean</ejb-class>
      <persistence-type>Container</persistence-type>
      <prim-key-class>java.lang.Integer</prim-key-class>
      <reentrant>false</reentrant>
```

```
      <cmp-version>2.x</cmp-version>
      <abstract-schema-name>Category</abstract-schema-name>
      <cmp-field id="CMPAttribute_1107874632507">
        <field-name>id</field-name>
      </cmp-field>
      <cmp-field id="CMPAttribute_1107874633048">
        <field-name>name</field-name>
      </cmp-field>
      <cmp-field id="CMPAttribute_1107874633068">
        <field-name>description</field-name>
      </cmp-field>
      <primkey-field>id</primkey-field>
      <query>
        <description></description>
        <query-method>
          <method-name>findAll</method-name>
          <method-params/>
        </query-method>
        <ejb-ql>select object(o) from Category o</ejb-ql>
      </query>
    </entity>
  ...
  </ejb-jar>
```

Following through with our design guidelines, we define a façade[5] object for the `Category` class. The goal of a session façade is to simplify the interface to do common tasks of a business entity. This often involves the coordination of methods in several objects. The session façade provides a simpler interface to find, create, modify, and delete entities. Fortunately for us, our development IDE provides automation to easily create session façades for entity beans. The resulting façade object is called `CategoryFacade` and, like the entity bean representation, defines functionality at both the remote and local levels and the instance and factory levels. In the model, a stereotyped «`facade`» dependency indicates the semantic connection between the design model–generated façade object and the design model representation of the entity. The façade class provides a number of utility methods for creating, finding, updating, and removing `Category` entities. This essentially acts as a single source manager of instances of this class. Figure 12–21 illustrates such a façade object.

Service Data Objects

Another characteristic of this particular façade implementation is the use of a relatively new and emerging standard, SDO. Simply put, SDO is a mechanism for accessing and manipulating data in a manner that is disconnected from the data

5. See the Session Façade pattern as described in *Core J2EE Patterns* [4].

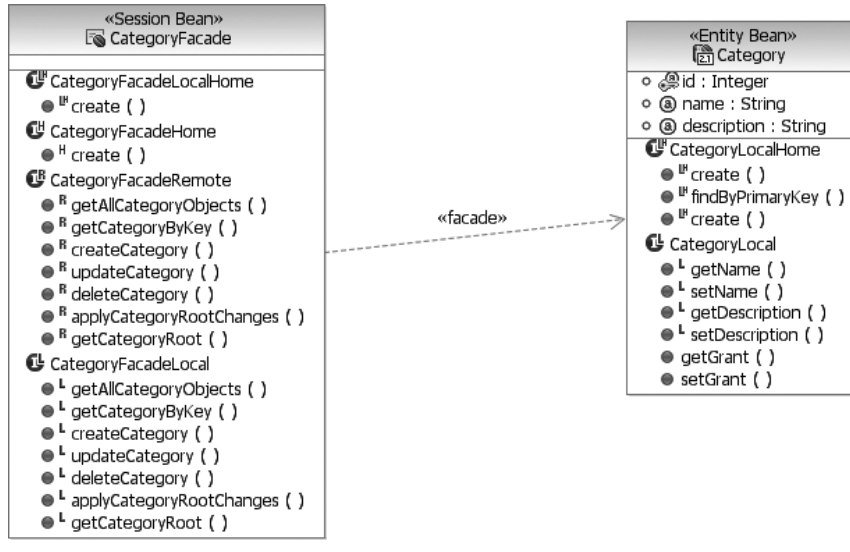

Figure 12–21 A Façade Object Governing Access to Entity Beans

source. This provides a useful means for marshalling data in a system. SDO uses disconnected data graphs and enables clients to retrieve the data graph from a source, manipulate it, and then apply the changes back to the original data source.

The use of SDO makes for a more consistent interface to bean data across the entire application. When the façade classes were generated, the corresponding set of SDO objects was also generated. Each SDO object that is created defines two interfaces: a root object (for the data graph) and an entity interface. The root object represents the conceptual root of the data graph. See Figure 12–22.

The façade object uses the SDO object as the main parameter for its create, retrieve, update, and delete (CRUD) methods. It also provides a method to update any changes made to an entire graph of SDO objects. Typical usage of this façade would be by another session bean acting as a business logic controller, perhaps fulfilling a request on behalf of a Web page. Let's take as a sample scenario the need to update and create vacation time request categories. The responsibility for this particular task falls on the HR clerk role. Such a business logic controller would need to be able to get all the existing categories (usually only a dozen or so)[6] to list on a Web page. It would then allow the user to select one for editing or

6. Arbitrary use of the `getAllObjects()` methods in the façade classes could lead to unexpected and serious performance issues when the system is tested in a real-life situation. Imagine the performance cost of putting into an array the collection of all the customer addresses in any medium-sized business.

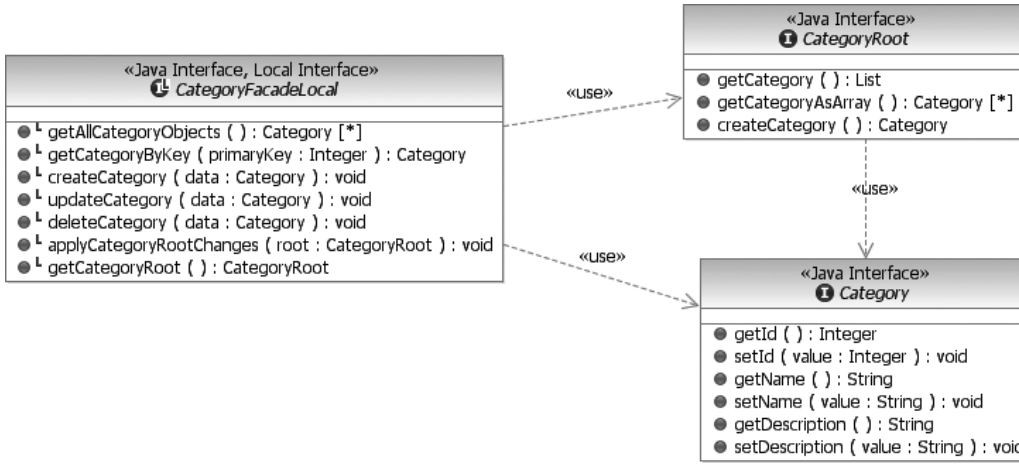

Figure 12–22 SDO Objects Used by the `CategoryFacadeLocal` Class

removal. The business logic controller can easily delete a category with a reference to the SDO or its primary key value. Changing the values of a category is similarly handled by changing the values in the SDO and then telling the façade to use its local value to update the persistent entity in the database.

Primary Key Generation

A pervasive issue in all object-relational database systems is the generation of primary keys for entity beans. The availability of natural primary key values like social security numbers or Universal Product Codes (UPCs) are highly dependent on the domain and can't always be assumed to be available. This means that the application must define and implement a strategy for creating primary keys for new entities that have no natural key attributes. In our system, the entities `Grant`, `Restriction`, and `Request` are examples of business objects that have no single attributes that could be safely used as primary key values.

In traditional database systems, a key could be manufactured as a composite of several properties and foreign key values; however, most EJB designs are more efficient with a single primary key value of a primitive type like integer or string. This means that we need to have a strategy for creating primary keys for our entities. Ideally, the same strategy and mechanism could be used for all our entity types that need generated key values.

In this application, we are have selected the Sequence Blocks strategy for primary key generation as described by Marinescu [5]. This strategy refines a very simple approach that essentially creates a new type of CMP entity responsible for man-

aging the next available integer value that can be used as a key for a particular type of entity. The primary problem with this approach is performance, especially in a system where new entities are created often. These performance issues are addressed by managing access to the entity with a session bean that reserves a large block of potential key values. Thus access to the CMP entity holding the keys is slightly reduced, depending on the size of keys reserved by the session beans.

Incorporating this strategy in our application leads to the creation of another entity and session bean with local access by the other entity façade objects (Figure 12–23). The `SequenceEntity` bean has only two attributes: the sequence name (i.e., the name of the entity type requiring a generated primary key) and the next available integer value that can be safely used as a primary key. Because this strategy reserves blocks of key values at a time, it is entirely likely and possible that there will be gaps in the sequence of integer values actually used. Thus, no application using this strategy should make any assumptions about the order or distribution of the primary key values.

The key generator is used by the other façade objects when there is a need to create a completely new instance of an entity. This is accomplished by updating the `createCategory()` method on the façade object to check for an incoming `Category` SDO with a null `id` field. If the `id` field is null, the façade is responsible for generating a fresh key for this category. Failure to do so will result in an exception being thrown.

Getting a new primary key value is as simple as getting access to the `SequenceSession` bean and then calling the `getNextSequenceNumber()` method with the name of the entity as a parameter. The code for the `createCategory` method is shown in Listing 12–2. The `doApplyChanges()` method is an SDO

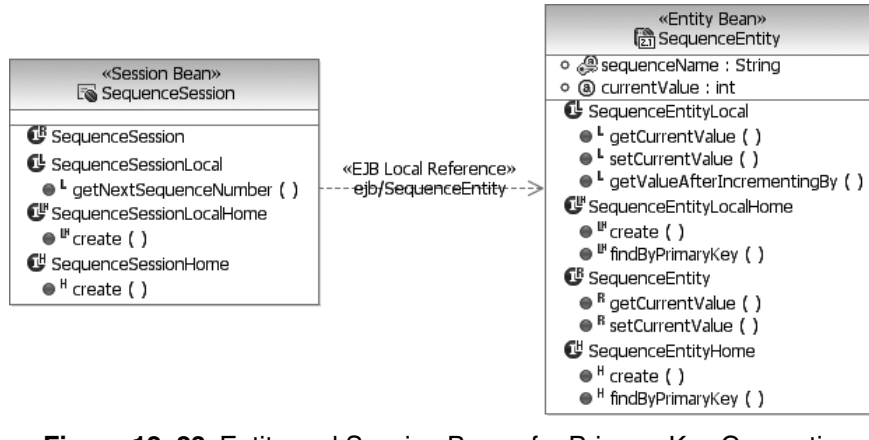

Figure 12–23 Entity and Session Beans for Primary Key Generation

framework implementation method, and generally speaking, the source code is unavailable.

Listing 12–2 Code for the *createCategory* Method

```
public void createCategory(Category data)
            throws CreateException {
    try {
      if( data.getId() == null ) {
        InitialContext ctx = new InitialContext();
        SequenceSessionLocalHome home =
             (SequenceSessionLocalHome)
              ctx.lookup("java:comp/env/ejb/SequenceSession");
        SequenceSessionLocal sequence = home.create();
        int id = sequence.getNextSequenceNumber("Category");
        data.setId( new Integer(id) );
      }
      doApplyChanges(data);
    } catch (Exception ex) {
      throw new CreateException(
          "System error while creating \"Category\".", ex);
    }
  }
```

With this update to all the façade's `create` methods and the use of the `SequenceSession` and `SequenceEntity` beans, new instances of entities can be created without the worry of deriving or computing the required primary key of each instance (Figure 12–24).

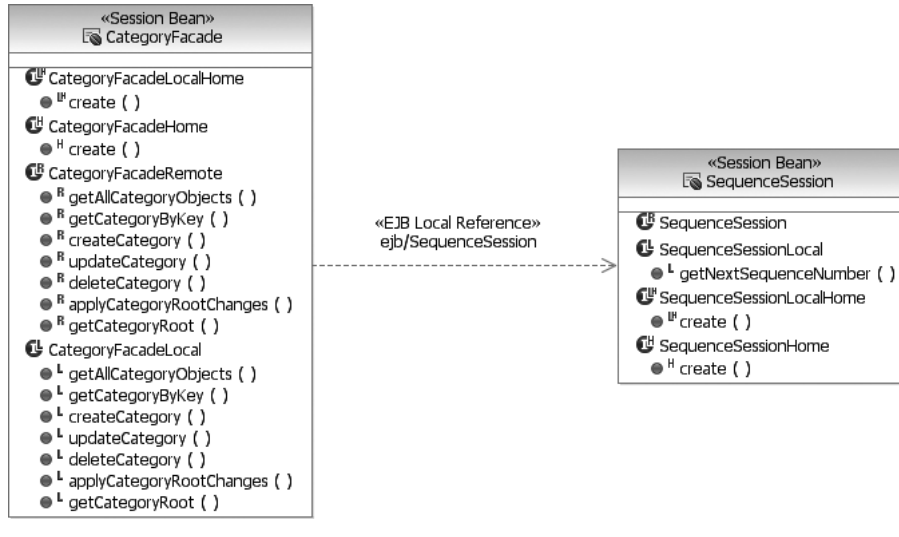

Figure 12–24 The `CategoryFacade` Session Bean Using the `SequenceSession` Bean to Create New Categories

Finders

Another important consideration in EJB entity design is the identification of finder methods. These are methods responsible for searching for specific entity instances that match a given set of criteria. The criteria can be as simple as all instances of vacation time request categories (which typically will result in a list of no more than a dozen instances). The finder may also implement a more complex search, for example, finding all employees whose requests for leaves of absence against a particular category of leave were rejected between March 1 and June 23 of the previous year. Regardless of details, the important point about finders is that the filtering of matching entities occurs on the server, ideally on the database server. Typically, this is the most efficient means of filtering large collections of entities. The only other option is to get all instances of an entity and then, after it has been potentially passed over the wire and reinstantiated in another process, iterate over all the instances and perform the matching tests. This is not the most efficient way to filter large collections.

The idea behind a finder is closely related to that of a SQL `SELECT` statement. EJB finders are expressed in a separate language, the EJB Query Language (EJB QL), which is very similar to SQL but not equivalent. For example, the following EJB QL statement will return all the `Grant` entity instances that are not associated with any restrictions.

```
select object(o) from Grant o where o.restrictions is not
empty
```

Finders can also be parameterized. In the next example, the finder query returns all `DateExclusionRestriction` instances that exclude the supplied date. Parameters are referenced in the query with a question mark followed by the parameter index in the argument list.

```
select object(o) from DateExclusionRestriction o where o.date
is ?1
```

Ultimately, these queries are captured in the main EJB configuration file `ejb-jar.xml` and associated with the Java method declared in the home interface.

Controllers

Even in Web applications there are many approaches to the concept of business logic control. Many Web-centric applications use the model-view-controller (MVC) approach. Currently the most popular framework for implementing strict MVC in J2EE Web applications is the Apache Struts framework. This framework

provides a small runtime component along with a set of JSP tag libraries for implementing the user interface and for making it easier to coordinate with the control framework. The MVC design pattern is a widely used and interpreted pattern. There are countless ways to implement the basic pattern even in the context of Web-centric applications. The Struts design and implementation was perhaps the first widely accepted implementation of this pattern for Java-based Web applications.

While the idea of implementing a strict MVC paradigm in the presentation tier sounds reasonable, on closer investigation of the system-level use cases, it is unclear where such control is necessary. In the overwhelming majority of use case scenarios, the functionality required is little more than basic CRUD operations on entities. Orchestrating and coordinating complex business operations, something for which an MVC paradigm is well suited, is just not part of our simple application.

The architect of this system has decided not to use an overt MVC paradigm like that supplied by Struts but instead to rely on the inherent coordination and control supplied as part of the Web pages. For some architects, the decision to not adopt an MVC and/or Struts control framework can be surprising, as many consider MVC critical to all Web-centric applications. We, however, consider the use of middleware, components, or even strategies that are not immediately necessary to the successful release of the application, or not obviously apparent in any existing documentation or future plans for the system, not to be worth the risk.[7]

With this decision made, all of the control constructs will be either in EJB session objects in the business tier, in beans in the server tier, or embedded as tags in the Web pages.

The Web Pages and the User Interface

The design of the Web pages is really separated into two major concerns: (1) the pages themselves, their hyperlinks, and their form fields and (2) their layout. For the first concern, the system architect and information architect collaborate to determine exactly what conceptual pages are required, what should be in them, and how they can be navigated through to accomplish the business goals of the

7. By this same reasoning, some might wonder why the decision was made to adopt EJBs for persistence, when it can be argued that a much simpler and more efficient persistence strategy could also be employed. This decision, like many made in a real-life development project, is often based as equally on technical merit as on organization history and experience.

system. Web pages are typically implemented as JSP pages and contain a mix of presentation data and relationships to business processing beans. The second concern, however, is all about aesthetics and user understanding. The layout of a Web page defines *where* information appears in the page, not *what* information should be there. It focuses on the organization of the page so that what the page presents can be most efficiently understood and worked with by the end users. The remainder of this chapter is devoted to the first concern, as the second one is more about art and visual creativity than traditional analysis and design.

Web page design begins with the UX model. The UX model more often than not describes a very good starting candidate for Web page names, content, and links. Generally speaking, UX «`screen`» classes map to individual JSP pages. Pure HTML Web pages can be part of a Web application when there is no dynamic content inside the page; however, even in these situations JSP pages are preferred since some client-side state management solutions require the dynamic rewriting of all URL hyperlinks, which can be done only with dynamic pages (see the Client State Management sidebar earlier in this chapter).

Depending on the Web tooling being used, the UX model can usually be easily transformed into a site layout or design model (Figure 12–25), where «`screen`» elements map to JSP pages and associations map to navigation links, either simple hyperlinks or forms where the user supplies additional data.

When designing the presentation tier, there are two main considerations: how to populate the page's dynamic content and how to invoke the business logic processes during page transitions. This application's architecture is J2EE, and the architect has specified the use of JSF to help integrate page construction with these two tasks. JSF helps simplify the construction of dynamic Web pages. It defines a number of custom tags that are embedded with HTML in JSP pages that are used to invoke methods on business objects and extract data to place in HTML elements that are eventually rendered on the client screen. In addition to the JSF tags, the Java Server Pages Standard Template Library (JSTL) also provides a number of useful tags to work with data in JSP pages.

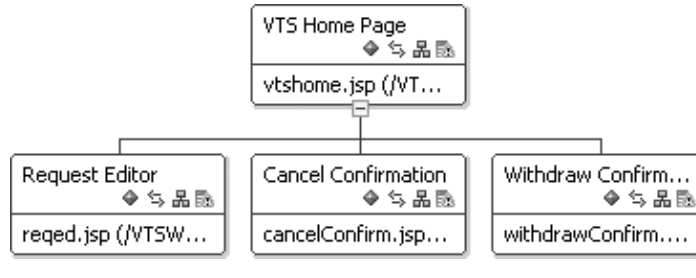

Figure 12–25 The Site Layout Represented in a Design Tool

Populating Dynamic Content

Populating a Web page with dynamic content requires a connection between a Java session bean in the presentation tier and a session bean in the business tier. This should be defined as a remote reference to enable the potential separation of tiers onto different nodes. To populate the VTS home page, we need, in addition to other information, the current employee summary of requests. This summary comes from an `Employee` CMP bean and its related `Request` CMP beans. Since access to CMP beans is governed by generated façade objects, the session bean in the presentation tier needs a remote reference to the façade object. The façade will return SDO objects to the presentation tier session bean, which in turn processes them to produce a simple Java object that can be used directly in the JSP.

In Figure 12–26, the `EmployeeSummarySession` class is a session bean that executes in the presentation tier. Its primary job is to connect to the

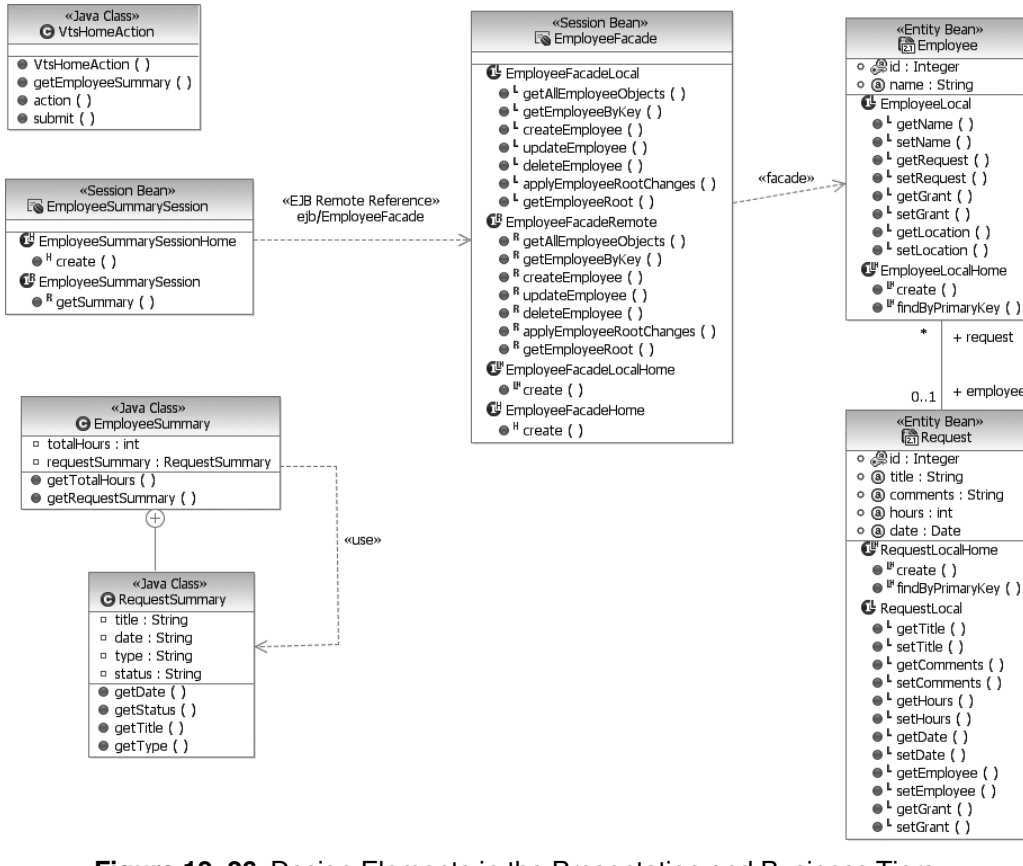

Figure 12–26 Design Elements in the Presentation and Business Tiers

EmployeeFacade and create a summary of an Employee object's requests. This session bean will also collect and make available other employee-specific information that will be used in the VTS home page. The EmployeeSummary class and its public inner class RequestSummary are POJOs used directly in the JSP pages via the JSTL and JSF tags. Part of the JSP source code for the VTS home page is shown in Listing 12–3.

Listing 12–3 JSP Source Code for the VTS Home Page

```
<HTML>
<HEAD>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f"%>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h"%>
<%@ page language="java" contentType="text/html;
 charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<jsp:useBean id="vtsHome" class="vtsweb.actions.VtsHomeAction"/>
...
<TITLE>VTS Home Page</TITLE>
</HEAD>
<f:view>
  <BODY>
  ...
  <P>Request Summary</P>

  <c:forEach items="${vtsHome.employeeSummary}">
    <c:url var="delUrl" value="faces/reqed">
       <c:param name="id" value="${leaveRequest.id}" />
       <c:param name="action" value="delete" />
    </c:url>
    <c:url var="edUrl" value="faces/reqed">
      <c:param name="id" value="${leaveRequest.id}" />
      <c:param name="action" value="edit" />
    </c:url>
    <tr>
      <td>${leaveRequest.date}</td>
      <td>${leaveRequest.type}</td>
      <td>${leaveRequest.status}</td>
      <td><a href="${delUrl}">delete</a></td>
      <td><a href="${edUrl}">edit</a></td>
    </tr>

  </c:forEach>

  <P><BR>Outstanding Grants</P>
  ...
  </BODY>
</f:view>
</HTML>
```

The VTS home page, like all of the JSP pages in our application, uses a number of custom tags in several different tag libraries. The three sets of tags that are used the most are the core JSF tags, the JSF HTML tags, and the JSTL tags. These are imported and associated with prefix identifiers at the beginning of the JSP source.

A JSP tag is used to bring in a reference to a presentation tier bean that is used for all access to business state and acts as a local controller for actions in the page. The bean is implemented as a POJO. It is referenced in the context of the JSP with the local reference `vtsHome`.

```
<jsp:useBean id="vtsHome" class=
    "vtsweb.actions.VtsHomeAction"/>
```

This bean can be used in JSF and JSTL custom tags to both extract business data and also invoke business methods. In our home page example, the JSTL `forEach` tag iterates over the `employeeSummary` collection of the bean, which contains instances of `RequestSummary` objects, each representing a specific vacation time request.

In general, it is a good practice to insulate JSP code and logic from the actual EJBs with a simple bean object that can be paired up with a specific page. This object is responsible for organizing and making available all the discrete values of data that can be placed in the JSP. This will also make it easier for the creative team members responsible for the final layout of the page, as they will not have to deal with the sometimes confusing technical requirements of managing EJBs.

Invoking Business Logic

Work is accomplished in Web applications primarily during page transitions. Logic is triggered by user requests to view or navigate to the next page. The next page may or may not be the actual page that the user requested. Depending on the outcome of the business logic, the desired page may get switched to a different page based on the current client's state. For example, if a user submits a request for vacation time but forgets to fill in a required field, the system will not return to the main home page as the user might expect but will return to the editing page and most likely display a warning message indicating the need for the required information.

In a JSF application, the navigation paths are defined by a configuration file called `faces-config.xml`. This file, among other things, defines the managed beans that can be referenced in JSP pages (e.g., `VtsHomeAction`). A navigation rule defines the allowed transitions from one page to another, based on an outcome value. This outcome value is computed within one of the managed beans, and when returned, the `faces` framework will use it to determine which

page to load and construct as a response. Listing 12–4 shows the navigation rules in `faces-config.xml`.

Listing 12–4 Navigation Rules in *`faces-config.xml`*

```
<navigation-rule>
  <from-view-id>/reqed.jsp</from-view-id>
  <navigation-case>
    <from-outcome>failure</from-outcome>
    <to-view-id>/reqed.jsp</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>success</from-outcome>
    <to-view-id>/vtshome.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
```

The rule in Listing 12–4 states that when the button or hyperlink component on `reqed.jsp` is activated, the application will navigate from the `reqed.jsp` page to the `vtshome.jsp` page if the outcome referenced by the button or hyperlink component's tag is `success`. Otherwise, the application will return to `reqed.jsp`.

Hyperlinks or form buttons are placed inside the originating page with the use of the JSF HTML tags. In the following command button example, clicking on the button will cause the `submit()` method of the `vtsHomeAction` instance to be invoked because the `action` attribute references the `submit` method of the `VtsHomeAction` backing bean. The `submit()` method performs some processing and returns a logical outcome that is passed to the default navigation handler, which matches the outcome against a set of navigation rules defined in the configuration file.

```
<h:commandButton value="Submit Changes"
action="#{vtsHome.submit}"/>
```

The impact of this design is that all of the presentation tier's logic is expressed in the managed or backing beans that execute in the Web server. These beans accomplish their tasks by creating normal remote EJB references to EJBs in the business tier, which could potentially be running on a different node in the system.

12.4 Transition and Post-Transition

Web-centric architectures, especially Internet-based ones, bring with them their own special implementation and testing concerns, in addition to the usual ones of

functionality and general performance. Web applications by their very nature exist in a heterogeneous environment that may be subject to change; the exact client hardware and software configuration can't be assumed. To help address these issues during design, special browser-specific technologies were purposely not used in our VTS application.

Implementing a Web application is, for the most part, the development of server-side software. Nearly all the software written in a typical Web application executes on the server side. Only when custom JavaScript is employed or specialized applets or client-side controls are developed will the developer really need to be concerned with the details of every client hardware and software configuration the application may encounter.

Even when you choose the technologies that are officially supported by the latest versions of the most popular browsers, there is no guarantee that they will perform the same way across browsers and client configurations. Such technologies and related issues include the following.

- *Java scripting*: All browsers do not implement client-side scripting in exactly the same way. Several browsers extend the scripting language with new and proprietary features. Others interpret the specifications slightly differently or have unusual side effects not governed by the official specifications.
- *Style sheets*: Style sheets and other layout-specific functionality depend on the availability of fonts and screen size. Just because you use style sheets doesn't mean that your pages will render in the same way on all client configurations.
- *Frames*: The implementation of frames has been problematic since their introduction. Scrolling, sizing, and targeting issues constantly plague the use of frames in applications. If frames are used in an application, be sure to test them completely, not only with all targeted client configurations but also in scenarios that involve more than just the use of the application being tested.
- *Bandwidth*: As our development tools become more sophisticated and it becomes easier to incorporate elaborate features into our Web pages, there is the chance that some end users will experience extremely slow page transitions due to the volume of HTML and/or the amount of scripting that must execute on the client. If the application under development is to be deployed to an anonymous user base over the Internet, special care must be taken to test and design the pages for low bandwidth and weak client configurations.

This set of issues becomes an ongoing challenge as technologies evolve over time.