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`

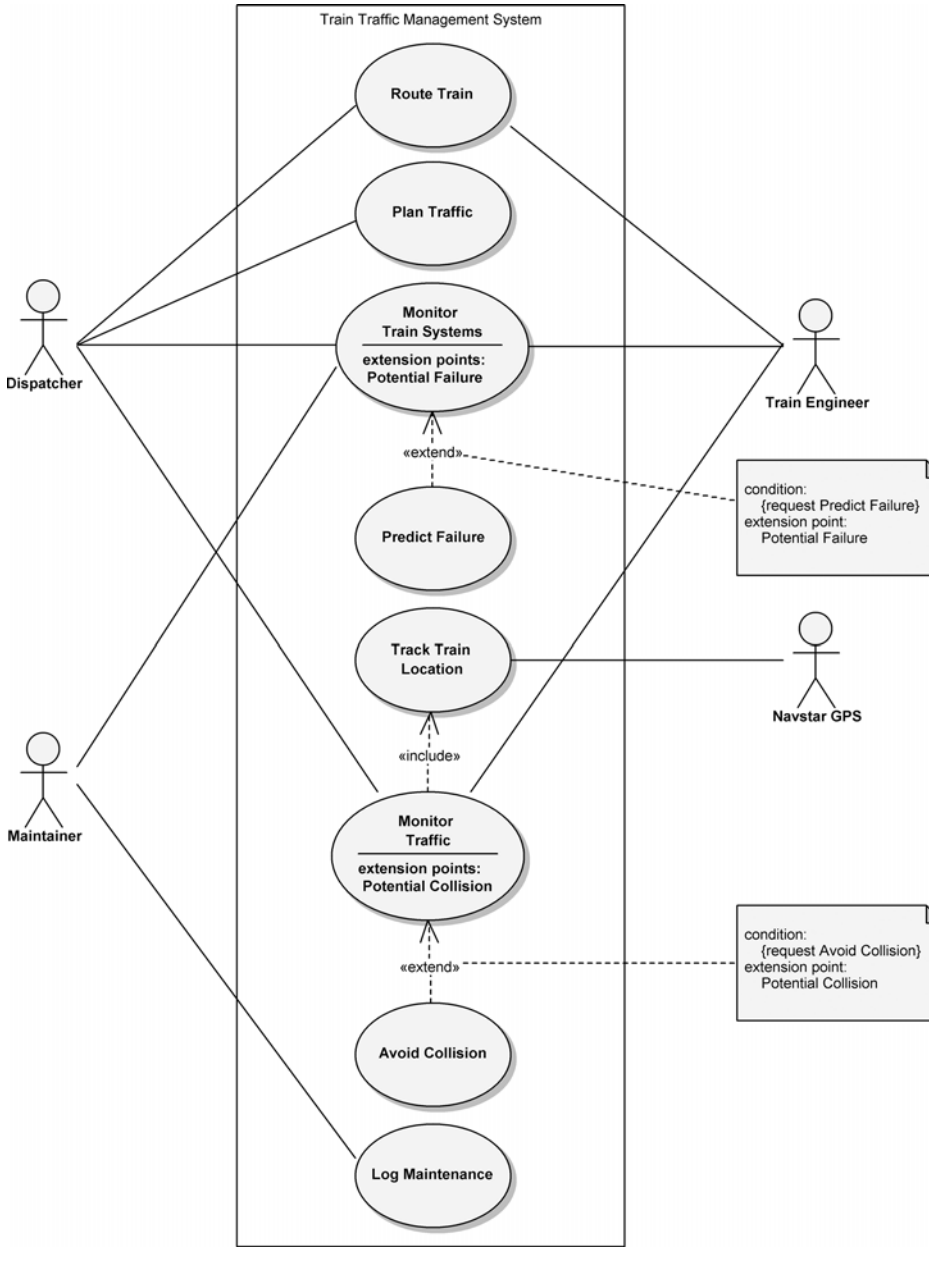

Figure 9–1 The Use Case Diagram for the Train Traffic Management System

`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.

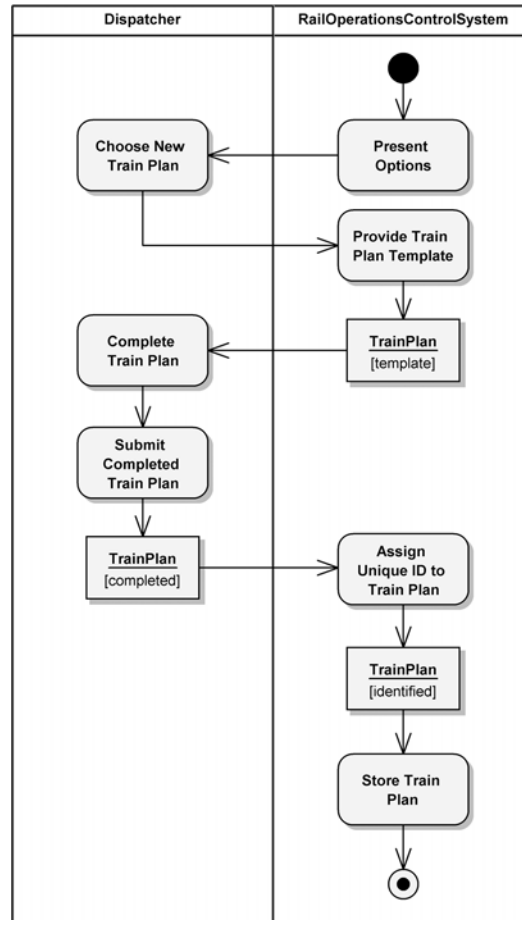

Figure 9–2 The `Route Train` Primary Scenario

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.

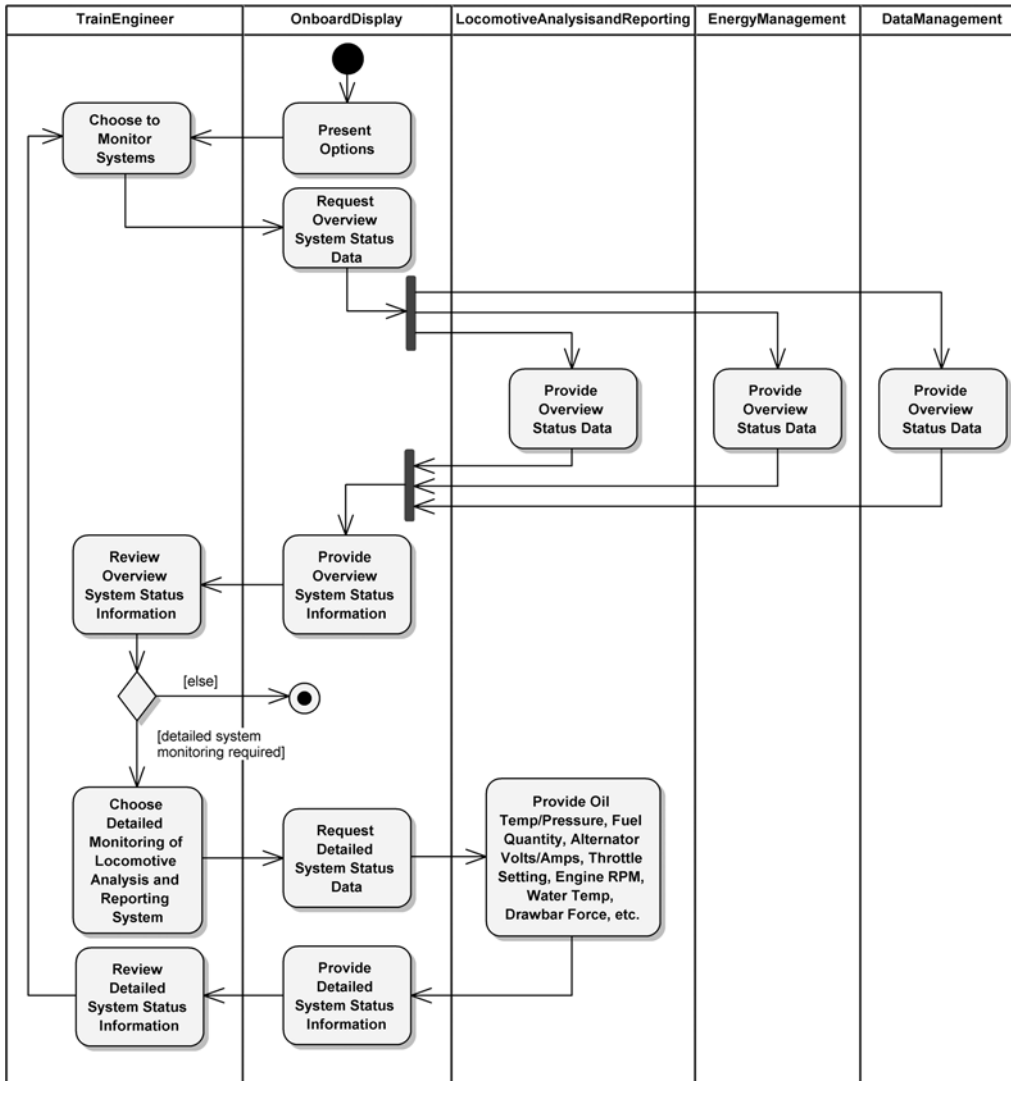

Figure 9–3 A `Monitor Train Systems` Alternate Scenario

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.

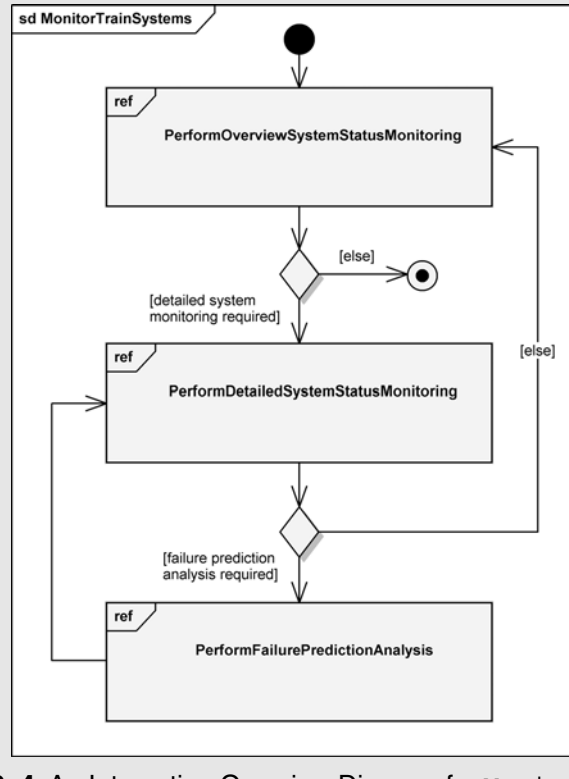

Figure 9–4 An Interaction Overview Diagram for `Monitor Train Systems`

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

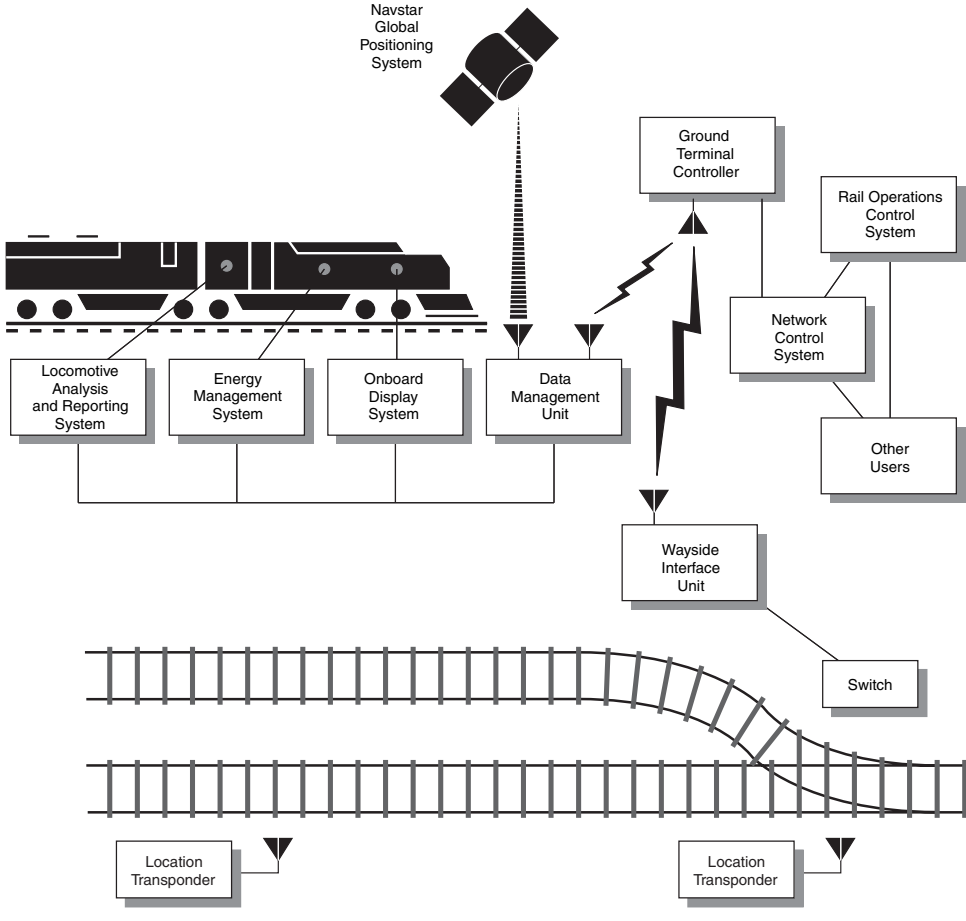

Figure 9–5 The Train Traffic Management System

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"

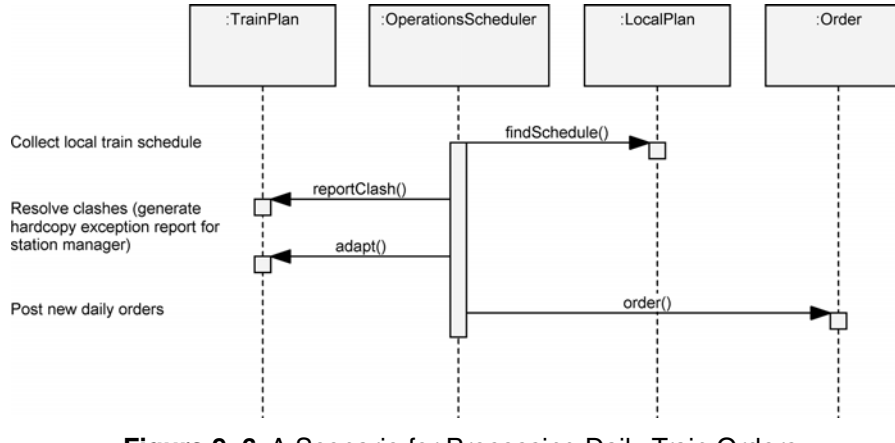

Figure 9–6 A Scenario for Processing Daily Train Orders

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.

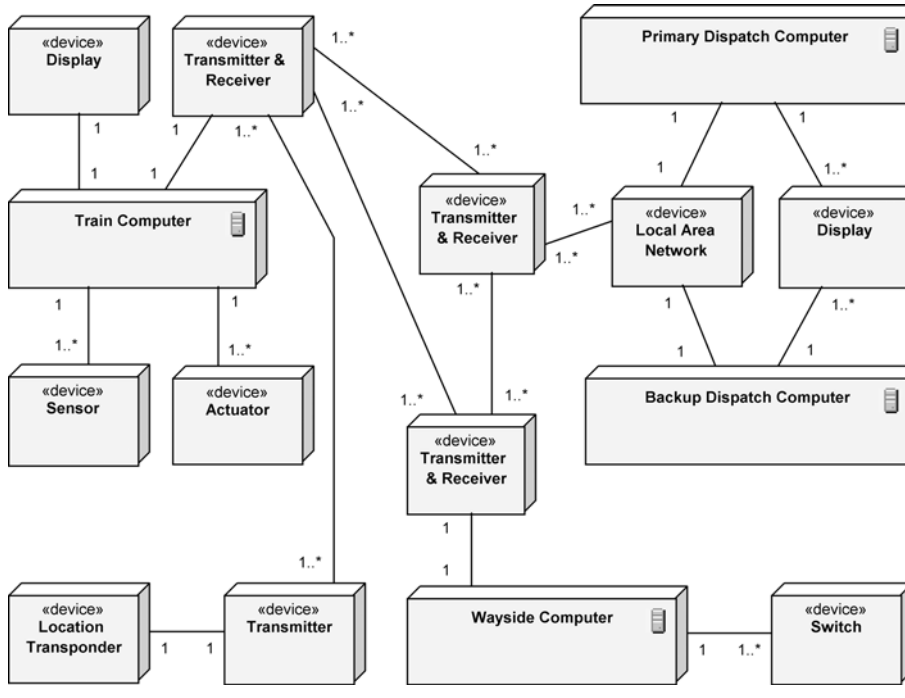

Figure 9–7 The Deployment Diagram for the Train Traffic Management System

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.

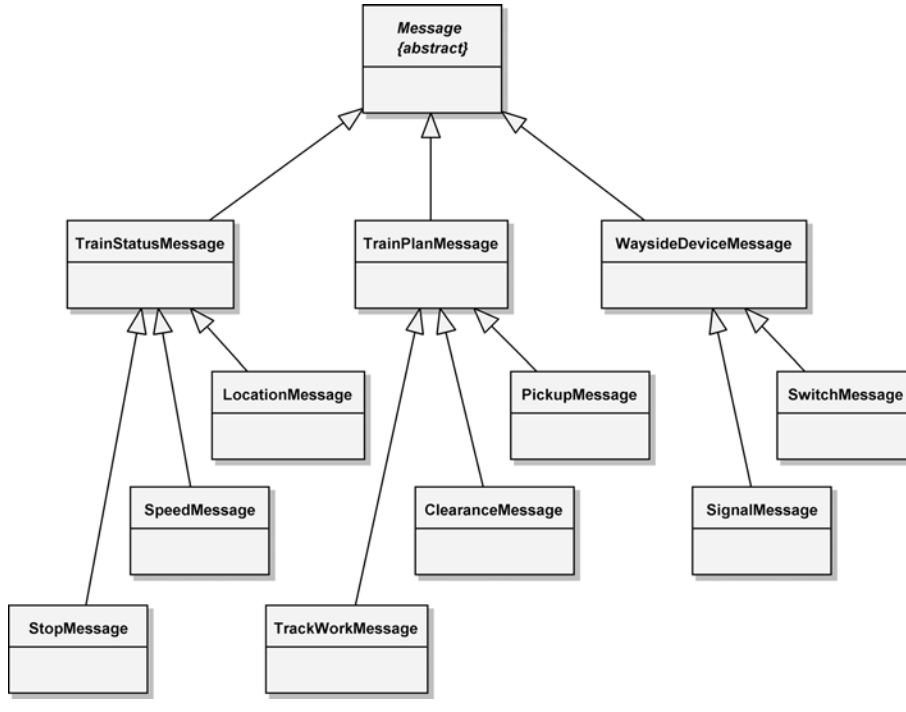

Figure 9–8 The `Message` Class Diagram

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

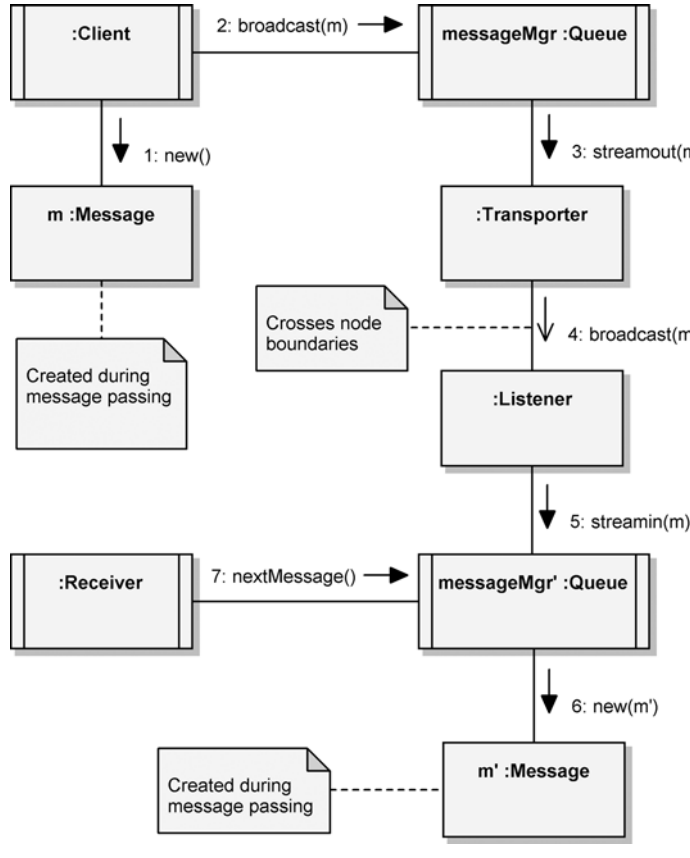

Figure 9–9 The Message-Passing Mechanism

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

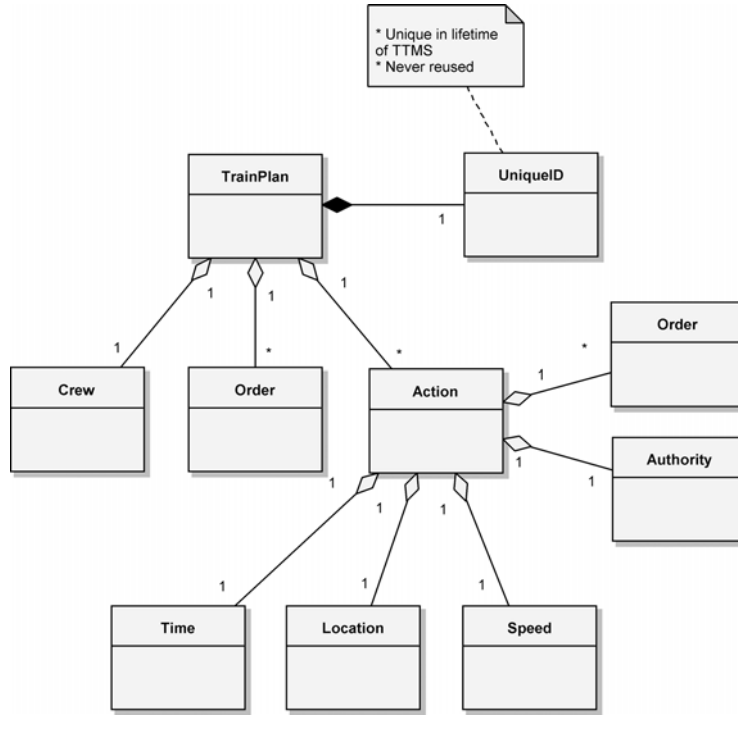

Figure 9–10 The `TrainPlan` Class Diagram

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

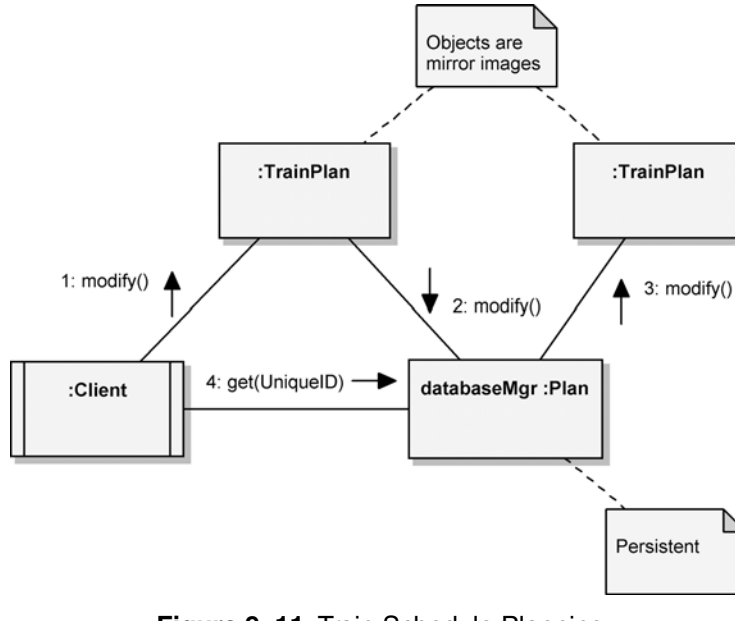

Figure 9–11 Train Schedule Planning

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-

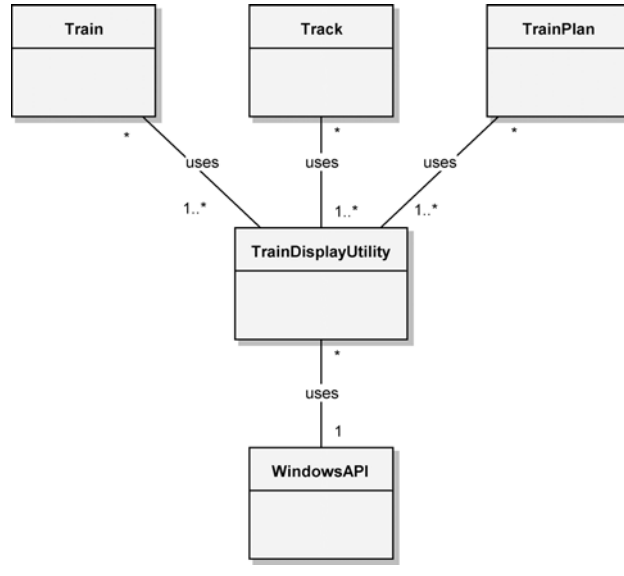

Figure 9–12 Class Utilities for Displaying

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

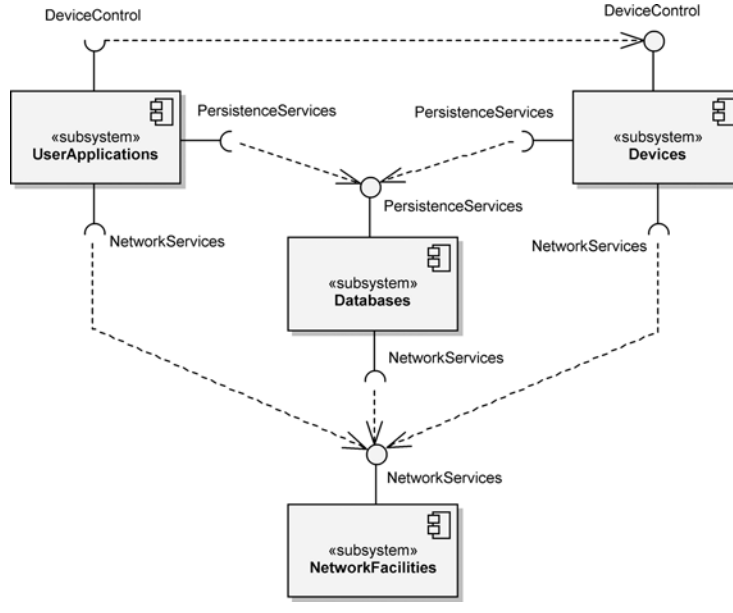

Figure 9–13 The Top-Level Component Diagram for the Train Traffic Management System

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.