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-

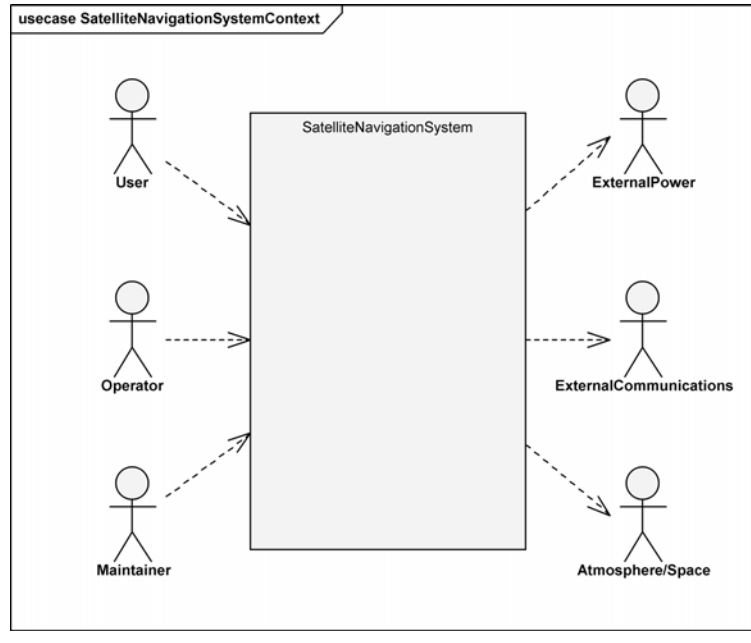

Figure 8–1 The Satellite Navigation System Context Diagram

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

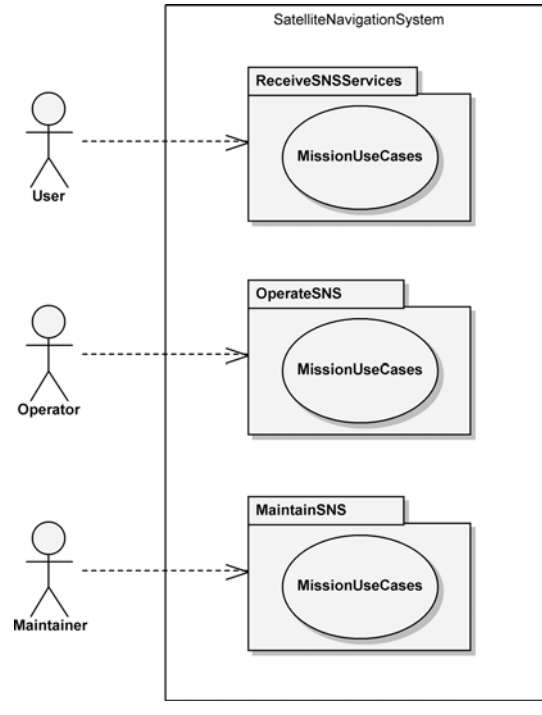

Figure 8–2 Packages for the SNS Mission Use Cases

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

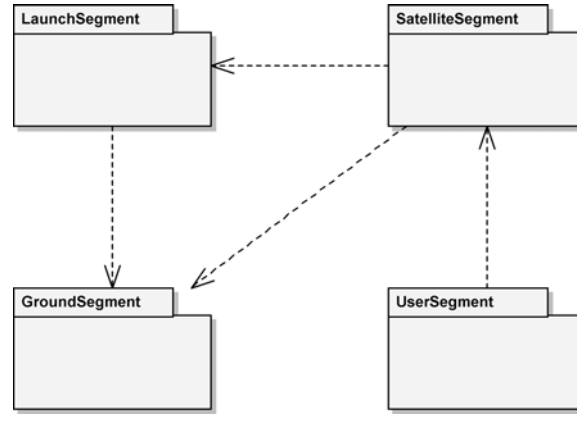

Figure 8–3 The SNS Logical Architecture

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

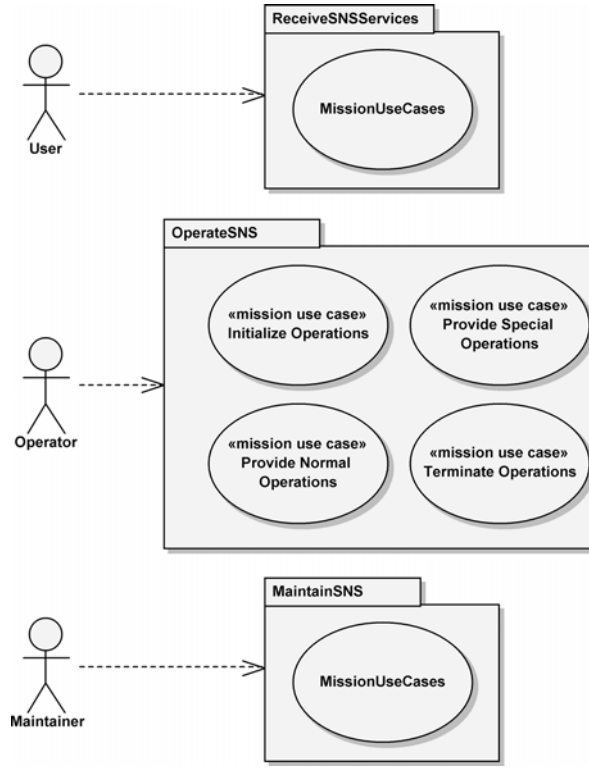

Figure 8–4 Refining the `OperateSNS` Mission Use Case Package

`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.

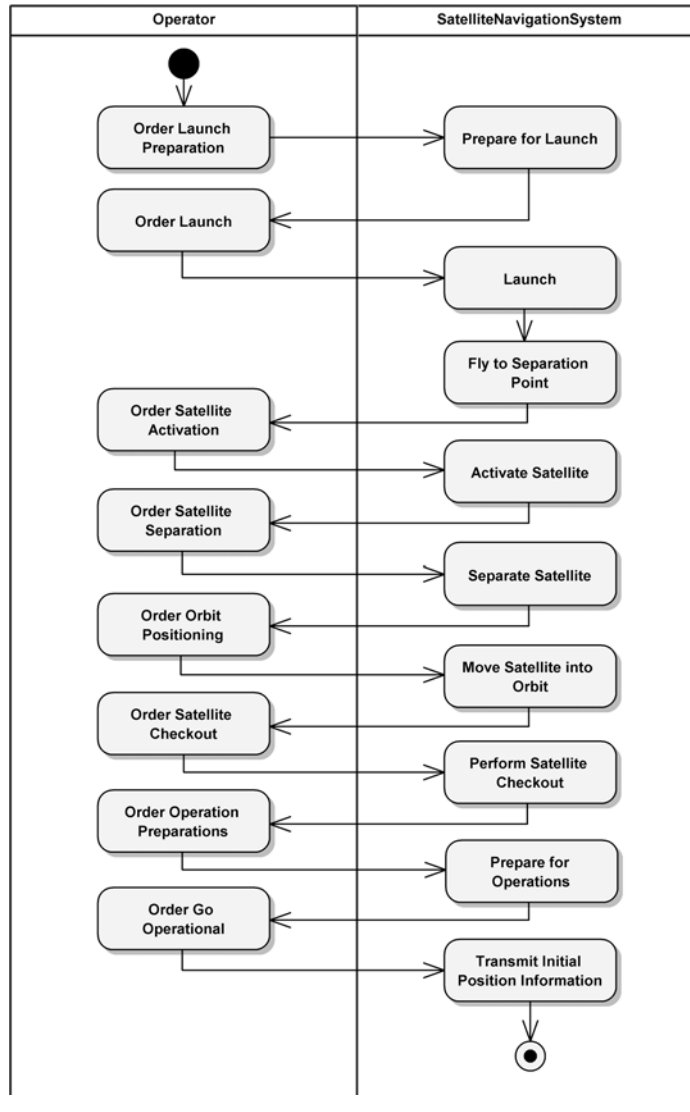

Figure 8–5 The Black-Box Activity Diagram for `Initialize Operations`

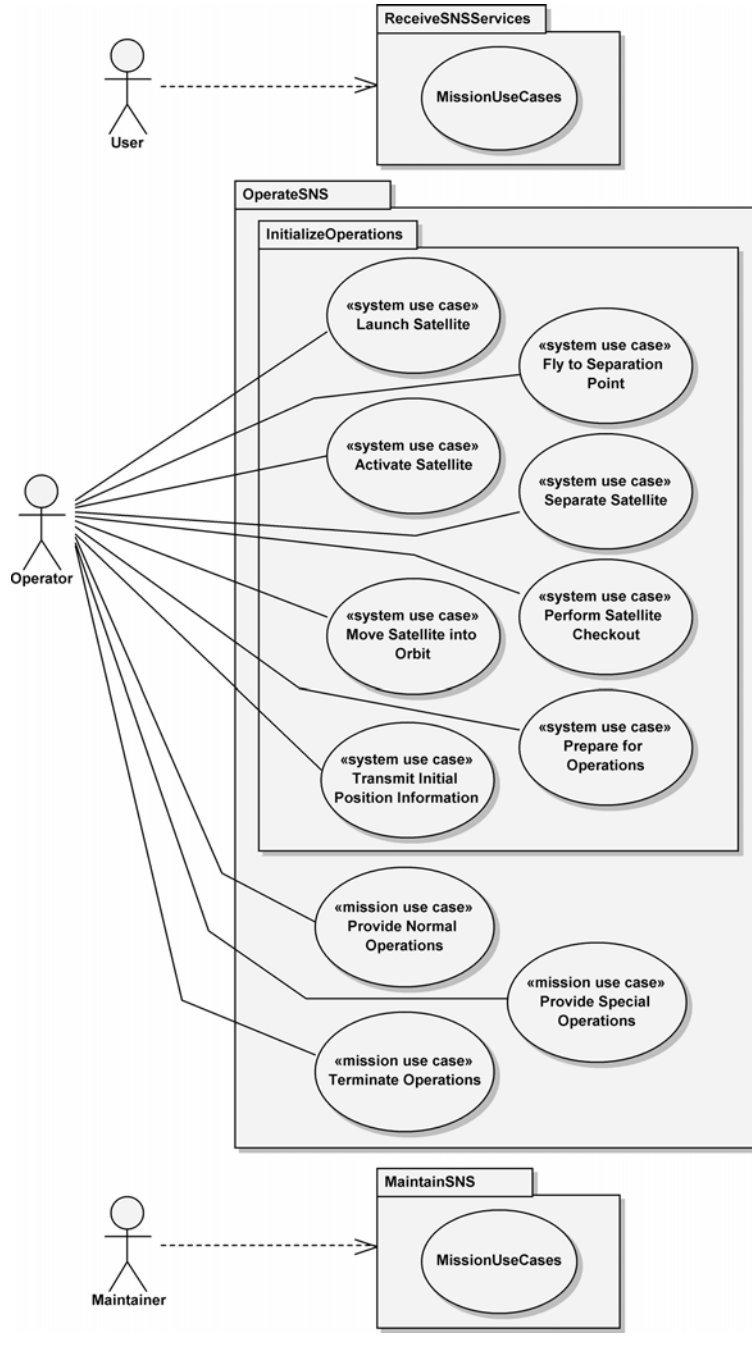

Figure 8–6 System Use Cases for `Initialize Operations`

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.

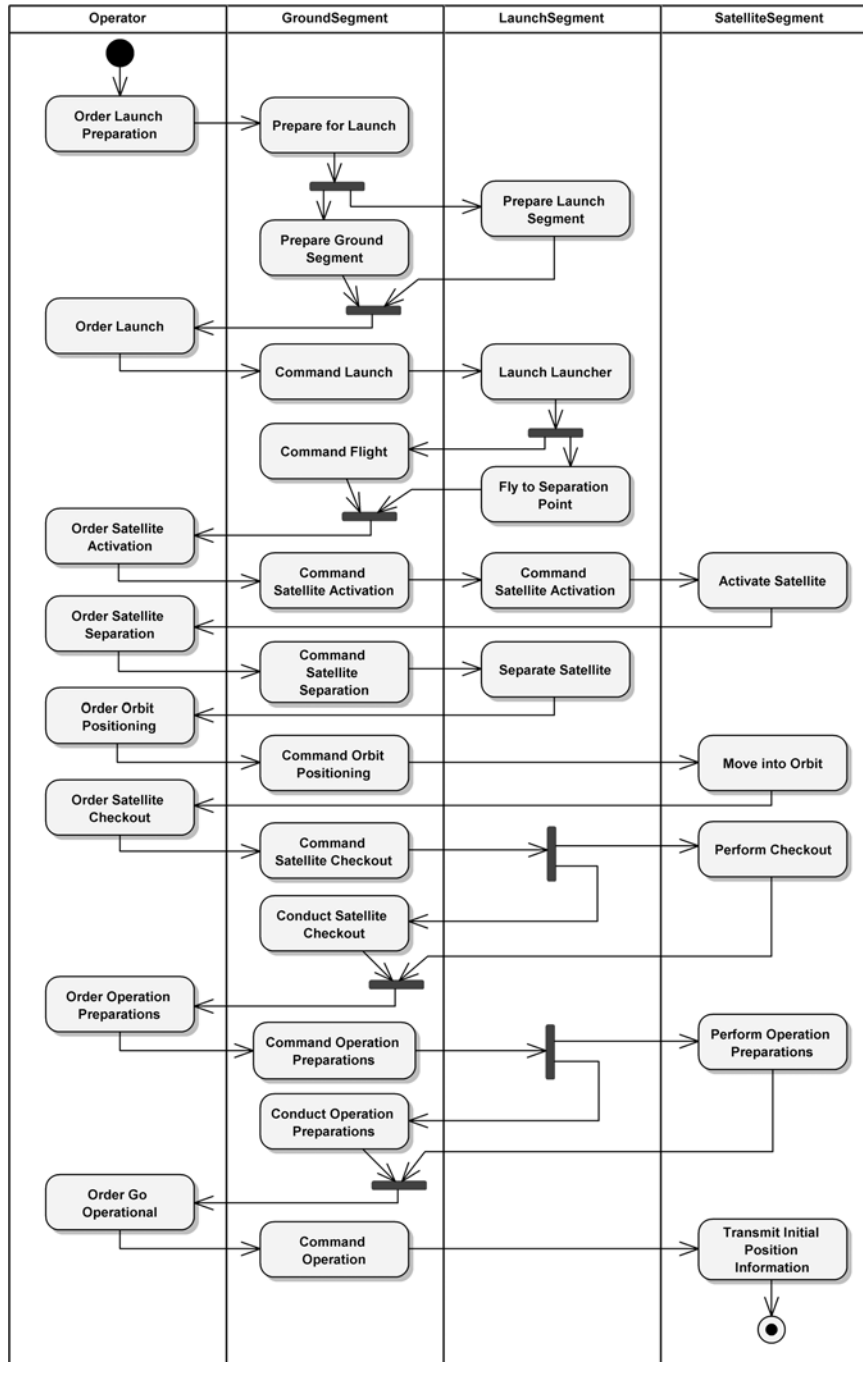

Figure 8–7 The White-Box Activity Diagram for `Initialize Operations`

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

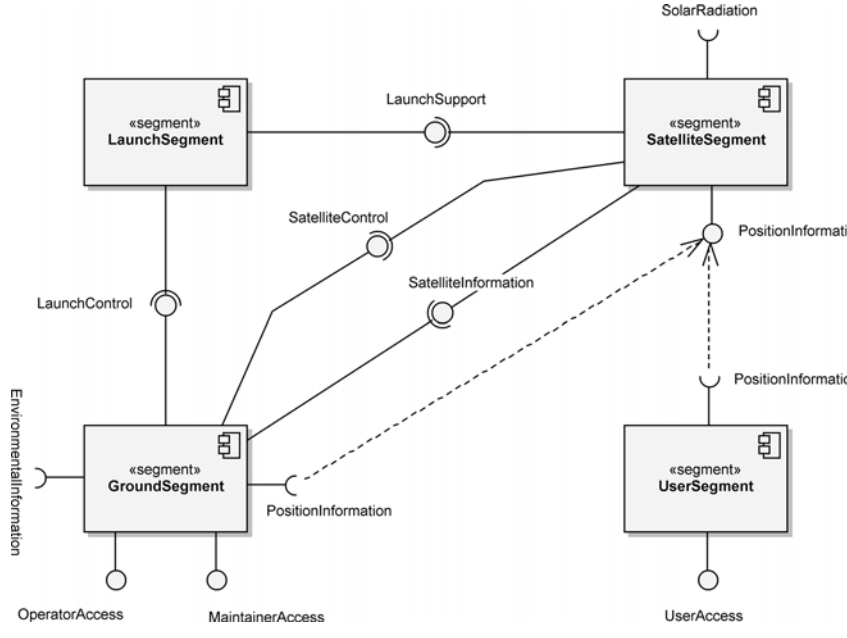

Figure 8–8 The Component Diagram for the Satellite Navigation System

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

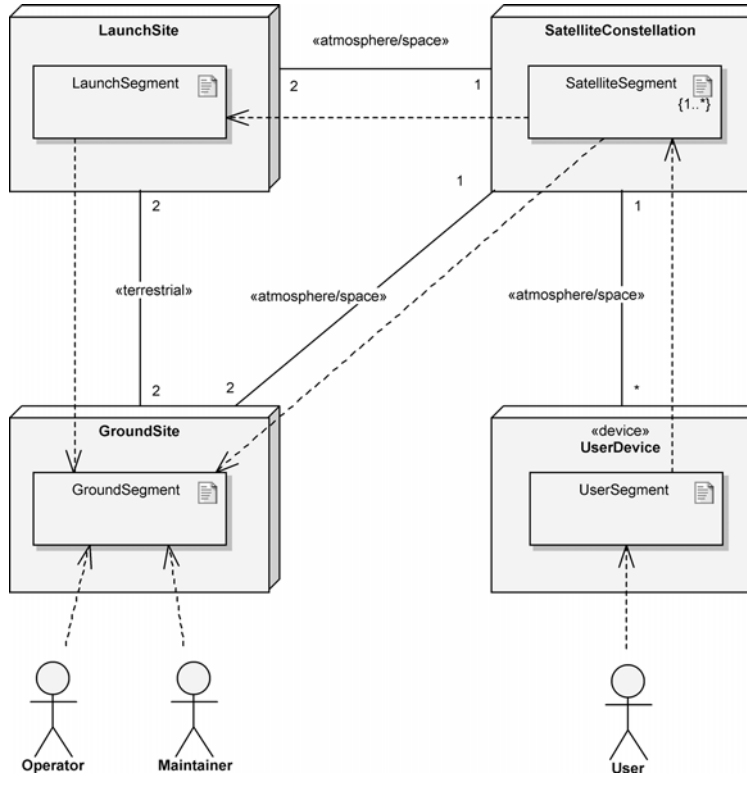

Figure 8–9 The Deployment of SNS Segments

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`.

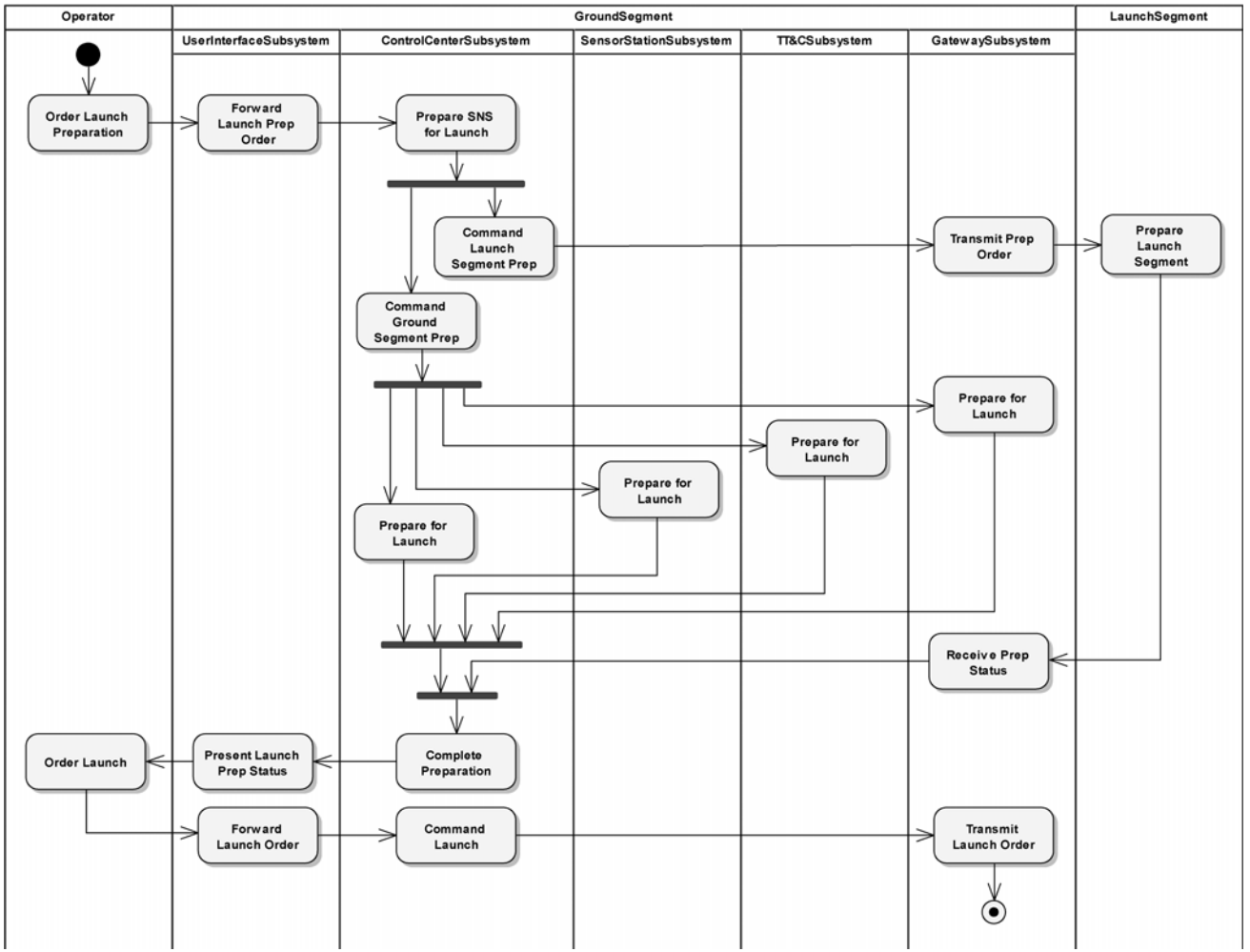

Figure 8–10 The White-Box Activity Diagram for `Control Launch`

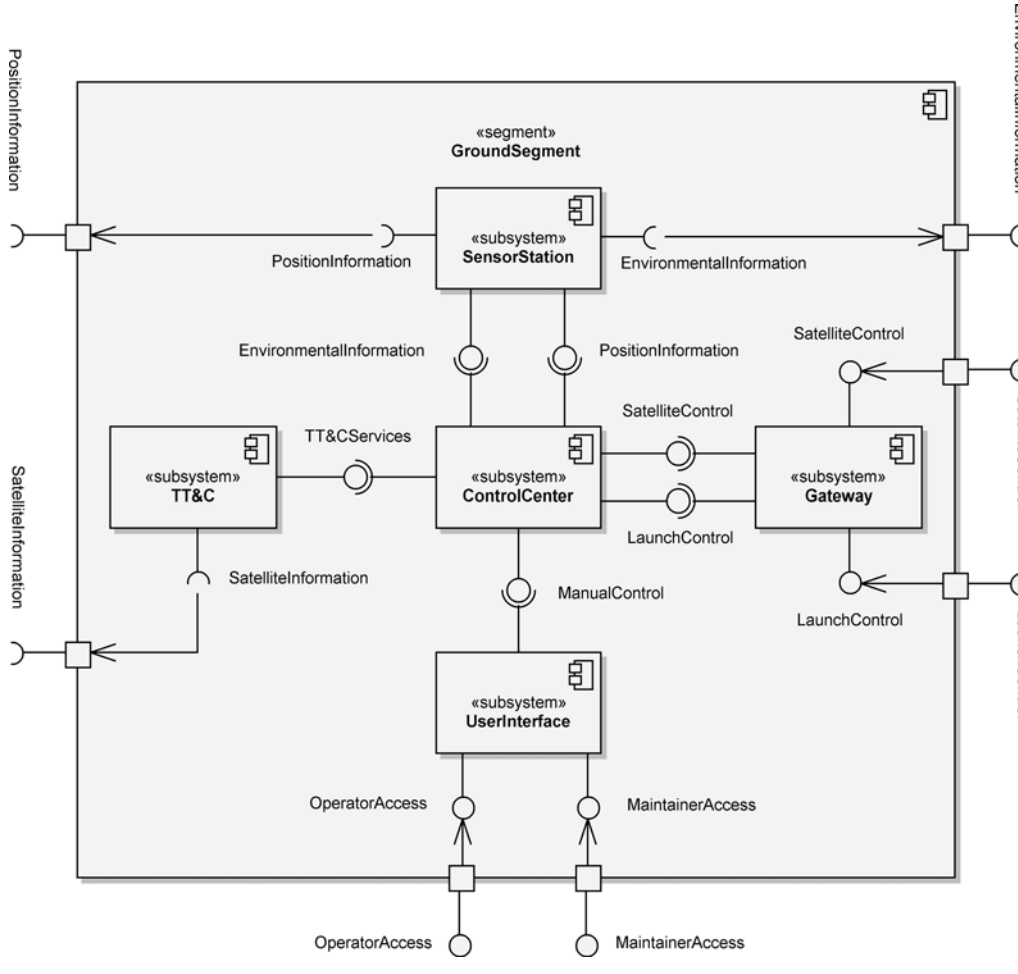

Figure 8–11 The Logical Architecture of the `GroundSegment`

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.

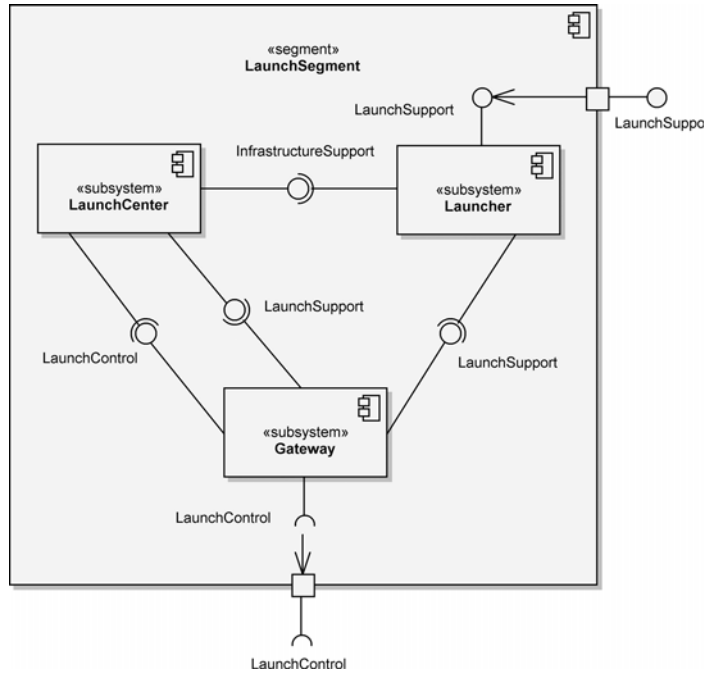

Figure 8–12 The Logical Architecture of the `LaunchSegment`

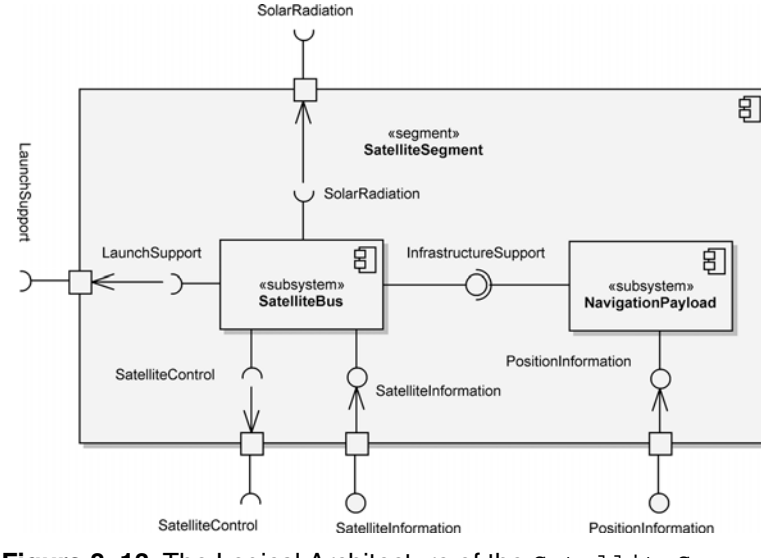

Figure 8–13 The Logical Architecture of the `SatelliteSegment`

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

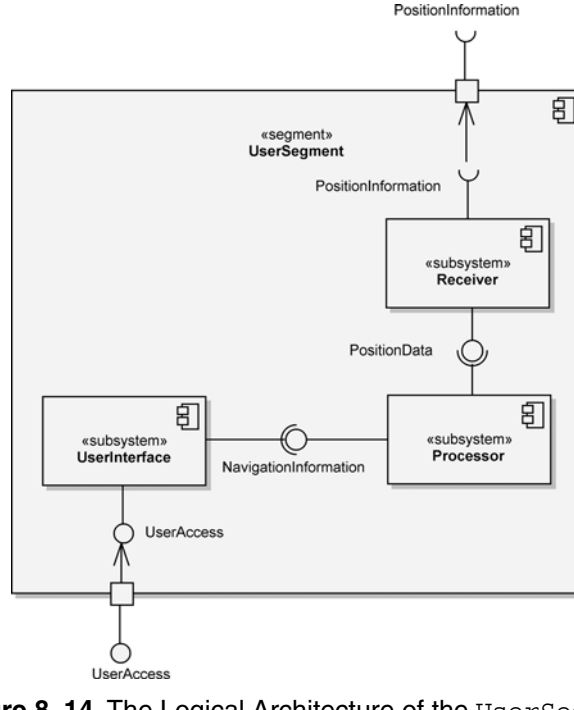

Figure 8–14 The Logical Architecture of the `UserSegment`

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).