OBJECT ORIENTED ANALYSIS AND DESIGN

(Professional Elective II)

UNIT I:

Introduction: The Structure of Complex systems, The Inherent Complexity of Software, Attributes of Complex System, Organized and Disorganized Complexity, Bringing Order to Chaos, Designing

Complex Systems. Case Study: System Architecture: Satellite-Based Navigation

UNIT II:

Introduction to UML: Importance of modeling, principles of modeling, object oriented modeling,

conceptual model of the UML, Architecture, and Software Development Life Cycle. Basic Structural Modeling: Classes, Relationships, common Mechanisms, and diagrams. Case Study: Control System: Traffic Management.

UNIT III:

Class & Object Diagrams: Terms, concepts, modeling techniques for Class & Object Diagrams.

Advanced Structural Modeling: Advanced classes, advanced relationships, Interfaces, Types and Roles, Packages. Case Study: AI: Cryptanalysis.

UNIT IV:

Basic Behavioral Modeling-I: Interactions, Interaction diagrams Use cases, Use case Diagrams, Activity Diagrams. Case Study: Web Application: Vacation Tracking System

UNIT V:

Advanced Behavioral Modeling: Events and signals, state machines, processes and Threads, time

and space, state chart diagrams. Architectural Modeling: Component, Deployment, Component diagrams and Deployment diagrams Case Study: Weather Forecasting

Text Books: 1. Grady BOOCH, Robert A. Maksimchuk, Michael W. ENGLE, Bobbi J. Young, Jim Conallen, Kellia Houston , “Object- Oriented Analysis and Design with Applications”, 3rd edition, 2013, PEARSON.

2. Grady Booch, James Rumbaugh, Ivar Jacobson: The Unified Modeling Language User Guide, Pearson Education.

UNIT I:

1. Introduction: The Structure of Complex systems.
2. The Inherent Complexity of Software.
3. Attributes of Complex System, Organized and Disorganized Complexity.
4. Bringing Order to Chaos.
5. Designing Complex Systems.
6. Case Study: System Architecture: Satellite-Based Navigation.

1.Introduction: The Structure of Complex systems.

- Object-orientation is what's referred to as a programming paradigm. It's not a language itself but a set of concepts that is supported by many languages.
- Object-oriented analysis and design (OOAD) is a technical approach for analyzing and designing an application, system, or business by applying object-oriented programming, as well as using visual modeling throughout the software development process to guide stakeholder communication and product quality.

Analysis focus on understanding, finding and describing concepts in the problem domain.

Analysis is

- An investigation of the **problem** rather than how a solution is defined
- Analysis specifies **what the system should do**
- Requirements analysis is focus on an investigation of the requirements

Design —Design focuses on **how to accomplish the objective of the system.**

Design emphasizes a conceptual solution that fulfills the requirements, rather than its implementation.

For example, a description of a database schema and software objects.

Analysis

- Focus on understanding the problem
- (Generalized) Behavior
- System structure
- Functional requirements and Some recognition for non-functional requirements
- A small model

Design

- Focus on understanding the solution
- Performance, Efficiency…
- Close to real code
- Object lifecycles
- Non-functional requirements in detail
- A large model

Systems: Systems are constructed by interconnecting components (Boundaries, Environments, Characters, Emergent Properties), which may well be systems in their own right. The larger the number of these components and relationships between them, higher will be the complexity of the overall system.

Complexity: Complexity depends on the number of the components embedded in them as well as the relationships and the interactions between these components which carry;

- Impossible for humans to comprehend fully
- Difficult to document and test
- Potentially inconsistent or incomplete
- Subject to change
- No fundamental laws to explain phenomena and approaches.

1.The structure of Complex Systems

The structure of personal computer, plants and animals, matter, social institutions are some examples of complex system.

1. The structure of personal computer
2. The structure of plants and animals
3. The structure of matter
4. The structure of social institutions

1.The structure of a Personal Computer:

- A personal computer is a device of moderate complexity. Major elements are CPU, monitor, keyboard and some secondary storage devices.
- CPU encompasses primary memory, an ALU, and a bus to which peripheral devices are attached.
- An ALU may be divided into registers which are constructed from NAND gates, inverters and so on.
- All are the hierarchical nature of a complex system.

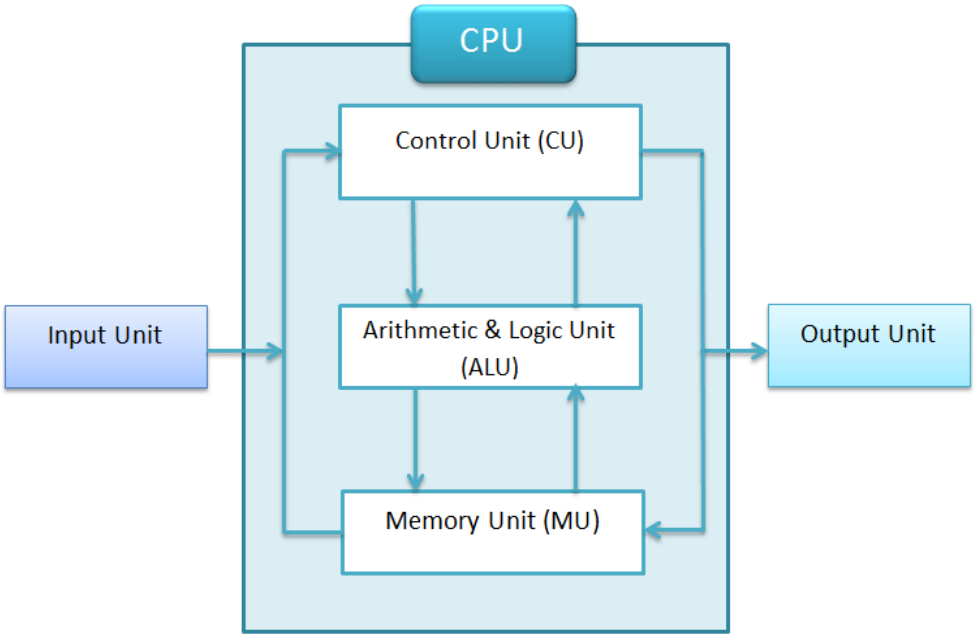
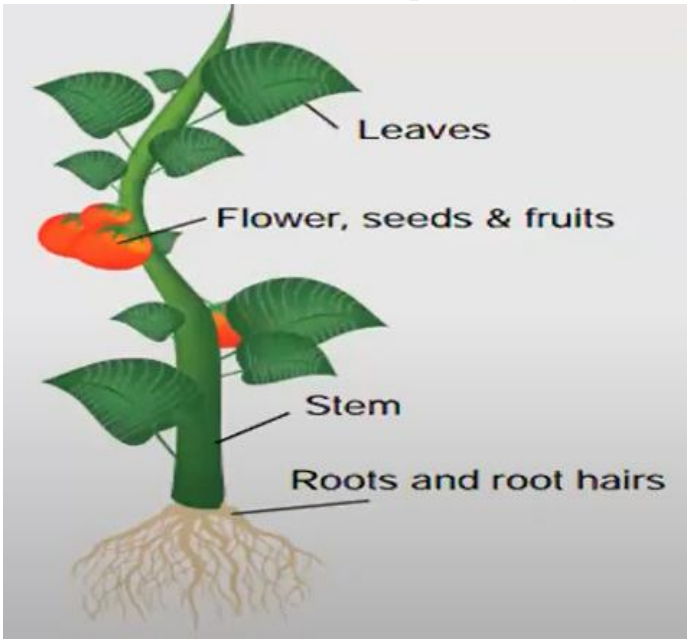


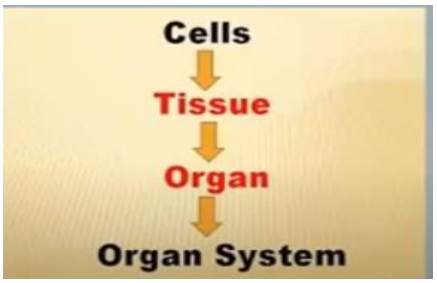
Fig: Block Diagram of Computer

2.The structure of Plants:

- Plants are complex multicellular organism which are composed of cells which is turn encompasses elements such as chloroplasts, nucleus, and so on.
- For example, at the highest level of abstraction, roots are responsible for absorbing water and minerals from the soil.
- Roots interact with stems, which transport these raw materials up to the leaves. The leaves in turn use water and minerals provided by stems to produce food through photosynthesis.
- All parts at the same level of abstraction interact in well-defined ways. for example, at the highest level of abstraction, roots are responsible for observing water and minerals from the soil.
- Roots interact with stems, which transport these raw materials up to leaves. The leaves in turn use the water and minerals provided by the stems to produce food through photosynthesis.

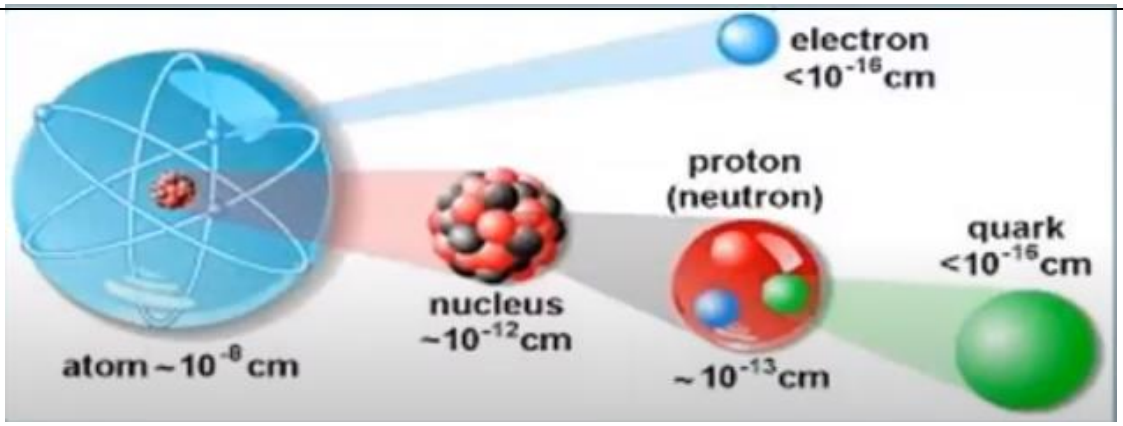


3.The structure of Animals: Animals exhibit a multicultural hierarchical structure in which collection of cells form tissues, tissues work together as organs, clusters of organs define systems (such as the digestive system) and so on.

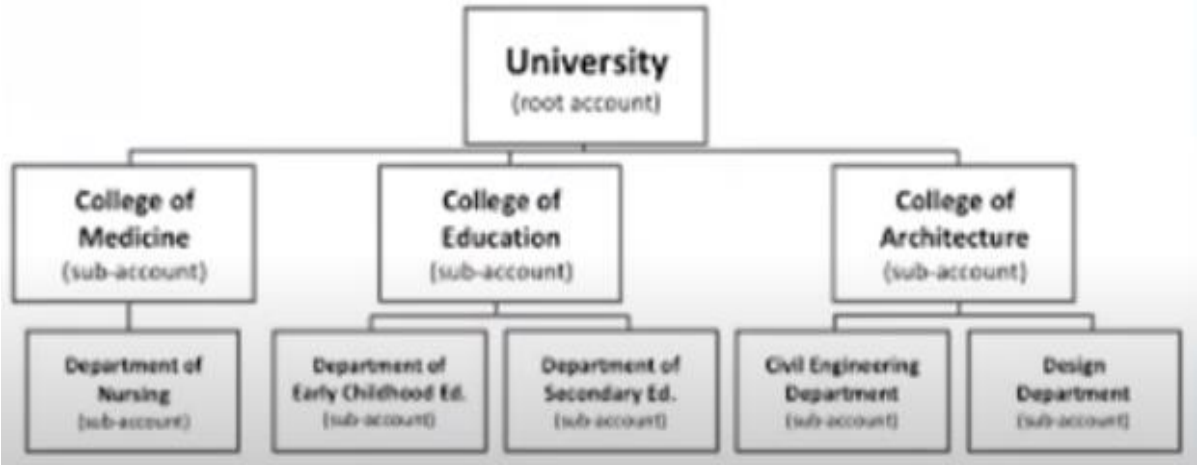


4.The structure of Matter:

- Nuclear physicists are concerned with a structural hierarchy of matter. Atoms are made up of electrons, protons and neutrons.
- Elements and elementary particles but protons, neutrons and other particles are formed from more basic components called quarks, which eventually formed from pro-quarks.



5.The structure of Social institutions: In social institutions, group of people join together to accomplish tasks that cannot be done by made of divisions which in turn contain branches which in turn encompass local offices and so on.



2.The Inherent Complexity of Software

The complexity of software is an essential property not an accidental one. The inherent complexity derives from four elements. They are

1. The complexity of the problem domain.
2. The difficultly of managing the developmental process.
3. The flexibility possible through software.
4. The problems of characterizing the behavior of discrete systems.

1.The complexity of the problem domain

1. The first reason has to do with the relationship between the application domains for which software systems are being constructed and the people who develop them.
2.Often, although software developers have the knowledge and skills required to develop software, they usually lack detailed knowledge of the application domain of such systems.

2. This affects their ability to understand and express accurately the requirements for the system to be built which come from the particular domain. Note, that these requirements are usually themselves subject to change.

3. They evolve during the construction of the system as well as after its delivery and thereby they impose a need for a continuous evolution of the system.

4. Complexity is often increased as a result of trying to preserve the investments made in legacy applications.

5. In such cases, the components which address new requirements have to be integrated with existing legacy applications.

6. This results into interoperability problems caused by the heterogeneity of the different components which introduce new complexities.

2.The Difficulty of Managing the Development Process

1. The second reason is the complexity of the software development process.
2. Complex software intensive systems cannot be developed by single individuals. They require teams of developers.
3. This adds extra overhead to the process since the developers have to communicate with each other about the intermediate artifacts they produce and make them interoperable with each other.
4. This complexity often gets even more difficult to handle if the teams do not work in one location but are geographically dispersed.
5. In such situations, the management of these processes becomes an important subtask on its own and they need to be kept as simple as possible.
6. None person can understand the system whose size is measured in hundreds of thousands, or even millions of lines of code.
7. Even if we decompose our implementation in meaningful ways, we still end up with hundreds and sometimes even thousand modules.
8. The amount of work demands that we use a team of developers and there are always significant challenges associated with team development more developers means more complex communication and hence more difficult coordination.

3.The flexibility possible through software

1. Software is flexible and expressive and thus encourages highly demanding requirements, which in turn lead to complex implementations which are difficult to assess.
2. The third reason is the danger of flexibility.
3. Flexibility leads to an attitude where developers develop system components themselves rather than purchasing them from somewhere else.
4. Unlike other industrial sectors, the production depth of the software industry is very large.
5. The construction or automobile industries largely rely on highly specialized suppliers providing parts.
6. The developers in these industries just produce the design, the part specifications and assemble the parts delivered.

7. **The software development is different**: most of the software companies develop every single component from scratch.
8. Flexibility also triggers more demanding requirements which make products even more complicated.
9. Software offers the ultimate flexibility. It is highly unusual for a construction firm to build an onsite steel mill to forge (create with hammer) custom girders (beams) for a new building.
10. Construction industry has standards for quality of raw materials, few such standards exist in the software industry

4.The problem of characterizing the behavior of discrete systems

1. The final reason for complexity according to Booch is related to the difficulty in describing the behavior of software systems.
2. Humans are capable of describing the static structure and properties of complex systems if they are properly decomposed, but have problems in describing their behavior.
3. This is because to describe behavior, it is not sufficient to list the properties of the system.
4. It is also necessary to describe the sequence of the values that these properties take over time.

3.The five Attributes of a complex system:

There are five attributes common to all complex systems. They are as follows:

1.Hierarchic Structure
2.Relative Primitives
3.Separation of Concerns
4.Common Patterns
5.Stable intermediate Forms

1.Hierarchic Structure

1. Frequently, complexity takes the form of a hierarchy, whereby a complex system is composed of interrelated subsystems that have in turn their own subsystems and so on, until some lowest level of elementary components is reached.
2. The fact that many complex systems have a nearly decomposable, and hierarchic structure is a major facilitating factor enabling us to understand, describe, and even "see" such systems their parts.

2.Relative Primitives: The choice of what components in a system are primitive is relatively arbitrary and is largely up to the discretion of the observer of the system class structure

and the object structure are not completely independent each object in object structure represents a specific instance of some class.

3.Separation of Concerns

1. Hierarchic systems decomposable because they can be divided into identifiable parts; he calls them nearly decomposable because their parts are not completely independent.
2. This leads to another attribute common to all complex systems.

3. Intra-component linkages are generally stronger than inter-component linkages.
4. This fact has the involving the high frequency dynamics of the components-involving the internal structure of the components – from the low frequency dynamic involving interaction among components.

4.Common Patterns

1. Hierarchic systems are usually composed of only a few different kinds of subsystems in various combinations and arrangements.
2. In other words, complex systems have common patterns.
3. These patterns may involve the reuse of small components such as the cells found in both plants or animals, or of larger structures, such as vascular systems, also found in both plants and animals.

5.Stable intermediate Forms

A complex system that works is invariably bound to have evolved from a simple system that worked ……. A complex system designed from scratch never works and can't be patched up to make it work. You have to start over, beginning with a working simple system.

4.Organized and Disorganized Complexity:

1. One mechanism to simplify concerns in order to make them more manageable is to identify and understand abstractions common to similar objects or activities.
2. Use a car as an example (which are considerable complex systems). Understanding common abstractions in this particular example would, for instance, involve the insight that clutch, accelerator and brakes facilitate the use of a wide range of devices, namely transport vehicles depending on transmission of power from engine to wheels.
3. Another principle to understand complex systems is the separation of concerns leading to multiple hierarchies that are orthogonal to each other. In the car example, this could be, for instance, the distinction between physical structure of the car (chassis, body, engine), functions the car performs (forward, back, turn) and control systems the car has (manual, mechanical, and electrical). In object- orientation, the class structure and the object structure relationship are the simplest form of related hierarchy.
4. It forms a canonical representation for object-oriented analysis.

The canonical form of a complex system

1. The discovery of common abstractions and mechanisms greatly facilitates are standing of complex system.
2. For example, if a pilot already knows how to fly a given aircraft, it is easier to know how to fly a similar one. May different hierarchies are present within the complex system.
3. For example, an aircraft may be studied by decomposing it into its propulsion system.
4. Flight control system and so on the decomposition represent a structural or "part of" hierarchy. The complex system also includes an "Is A" hierarchy.

5. These hierodules for class structure and object structure combining the concept of the class and object structure together with the five attributes of complex system, we find that virtually all complex

6.system take on the same (canonical) form as shown in figure. There are two orthogonal hierarchies of system, its class structure and the object structure.

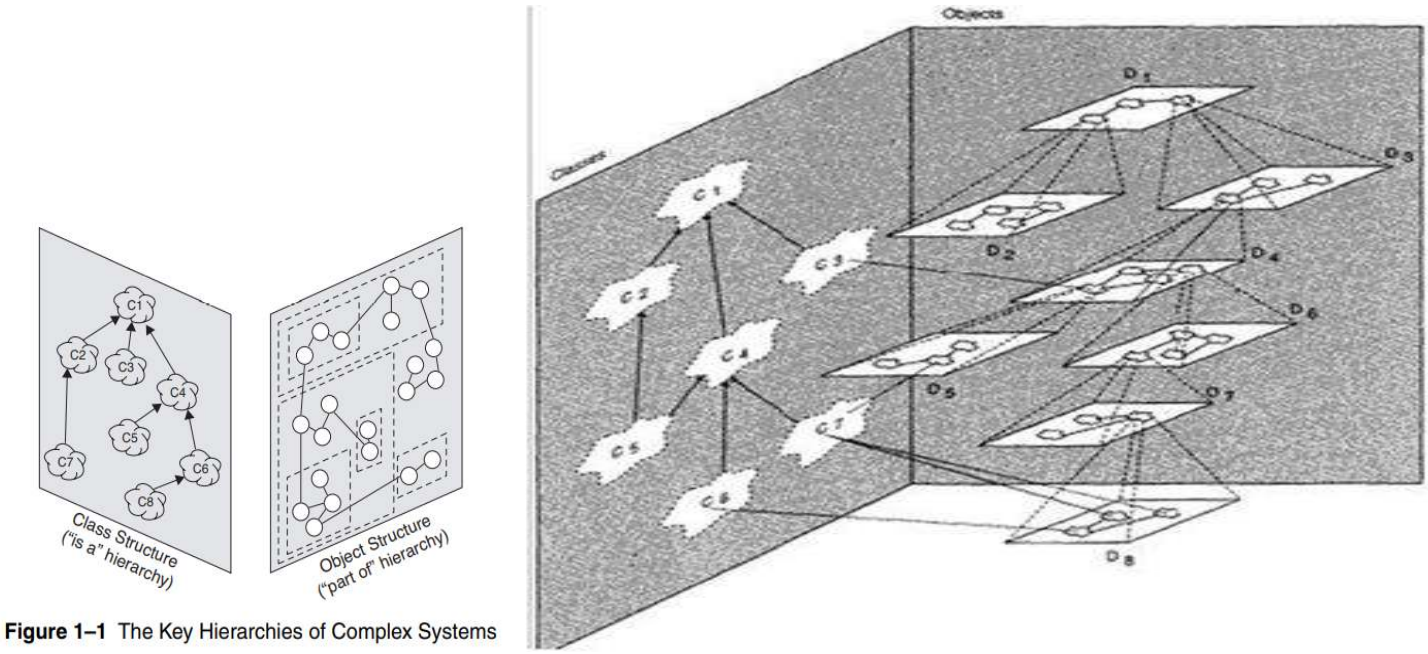


Figure 1–1 The Key Hierarchies of Complex Systems

Figure: Canonical form of a complex system

1. The figure represents the relationship between **two different hierarchies**:
2. A hierarchy of objects and a hierarchy of classes. The class structure defines the 'is-a' hierarchy, identifying the commonalities between different classes at different levels of abstractions.
3. Hence class C4 is also a class C1 and therefore has every single property that C1 has. C4, however, may have more specific properties that C1 does not have; hence the distinction between C1 and C4.
4. The object structure defines the 'part- of' representation.
5. This identifies the composition of an object from component objects, like a car is composed from wheels, a steering wheel, a chassis and an engine.
6. The two hierarchies are not entirely orthogonal as objects are instances of certain classes.
7. The relationship between these two hierarchies is shown by identifying the instance-of relationship as well.
8. The objects in component D8 are instances of C6 and C7 as suggested by the diagram, there are many more objects then there are classes.
9. The point in identifying classes is therefore to have a vehicle to describe only once all properties that all instances of the class have.

The Limitations of the human capacity for dealing with complexity:

1.Object model is the organized complexity of software.

2.As we begin to analyze a complex software system, find many parts that must interact in a multitude of intricate ways with little commonality among either the parts or their interactions. This is an example of disorganized complexity.

3.In complex system, find many parts that must interact in a multitude of intricate ways with little commonality among either the parts or their intricate.

4.This is an example in an air traffic control system, we must deal with states of different aircraft at once, and involving such it is absolutely impossible for a single person to keep track of all these details at once.

5.Bringing Order to chaos:

Principles that will provide basis for development

1. Abstraction
2. Hierarchy
3. Decomposition

1.The Role of Abstraction:

1.Abstraction is an exceptionally powerful technique for dealing with complexity.

2.Unable to master the entirely of a complex object, we choose to ignore its inessential details, dealing instead with the generalized, idealized model of the object.

3.For example, when studying about how photosynthesis works in a plant, can focus upon the chemical reactions in certain cells in a leaf and ignore all other parts such as roots and stems. Objects are abstractions of entities in the real world.

4.In general abstraction assists people's understanding by grouping, generalizing and chunking information.

5.Object- orientation attempts to deploy abstraction.

6.The common properties of similar objects are defined in an abstract way in terms of a class. Properties that different classes have in common are identified in more abstract classes and then an ‘is-a’ relationship defines the inheritance between these classes.

2.The role of Hierarchy:

1.Identifying the hierarchies within a complex software system makes understanding of the system very simple.

2.The object structure is important because it illustrates how different objects collaborate with one another through pattern of interaction (called mechanisms).

3.By classifying objects into groups of related abstractions (for example, kinds of plant cells versus animal cells, we come to explicitly distinguish the common and distinct properties of different objects, which helps to master their inherent complexity.

4.Different hierarchies support the recognition of higher and lower orders. A class high in the 'is-a' hierarchy is a rather abstract concept and a class that is a leaf represents a fairly concrete concept.

5.The 'is- a' hierarchy also identifies concepts, such as attributes or operations, that are common to a number of classes and instances thereof.

6.Similarly, an object that is up in the part-of hierarchy represents a rather coarse-grained and complex objects, assembled from a number of objects, while objects that are leafs are rather fine grained.

7.But note that there are many other forms of patterns which are nonhierarchical: interactions, relationships.

3.The role of Decomposition:

1.Decomposition is important techniques for copying with complexity based on the idea of divide and conquer.

2.In dividing a problem into a sub problem, the problem becomes less complex and easier to overlook and to deal with.

3.Repeatedly dividing a problem will eventually lead to sub problems that are small enough so that they can be conquered.

4.After all the sub problems have been conquered and solutions to them have been found, the solutions need to be composed in order to obtain the solution of the whole problem.

5.The history of computing has seen two forms of decomposition, process-oriented (Algorithmic) and object-oriented decomposition.

1.Algorithmic (Process Oriented) Decomposition: In Algorithmic decomposition, each module in the system denotes a major step in some overall process.

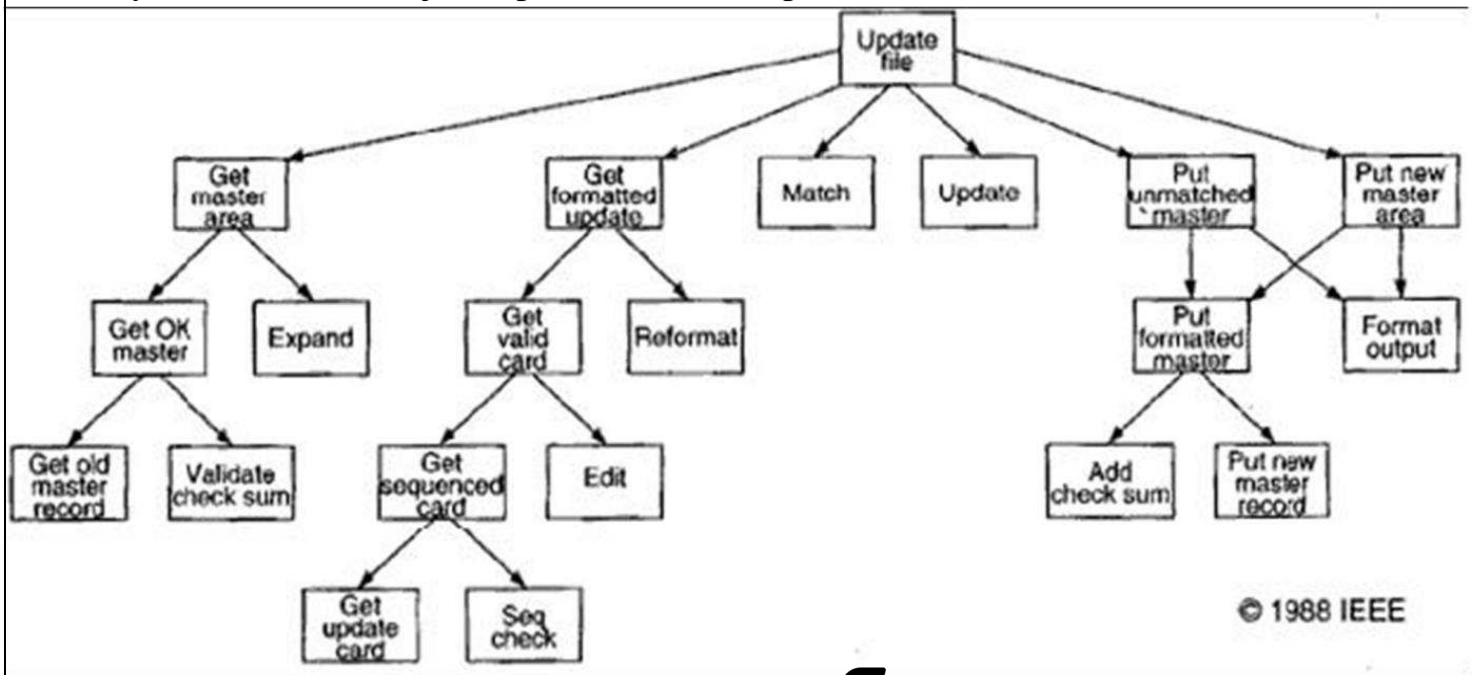


Figure: Algorithmic decomposition

2.Object oriented decomposition:

1. Objects are identified as Master file and check sum which derive directly from the vocabulary of the problem as shown in figure.
2. The world as a set of autonomous agents that collaborate to perform some higher-level behavior.
3. Get formatted update thus does not exist as an independent algorithm; rather it is an operation associated with the object file of updates.
4. Calling this operation creates another object, update to card. In this manner, each object in our solution embodies its own unique behavior.
5. Each hierarchy in layered with the more abstract classes and objects built upon more primitive ones especially among the parts of the object structure, object in the real world.
6. Here decomposition is based on objects and not algorithms.

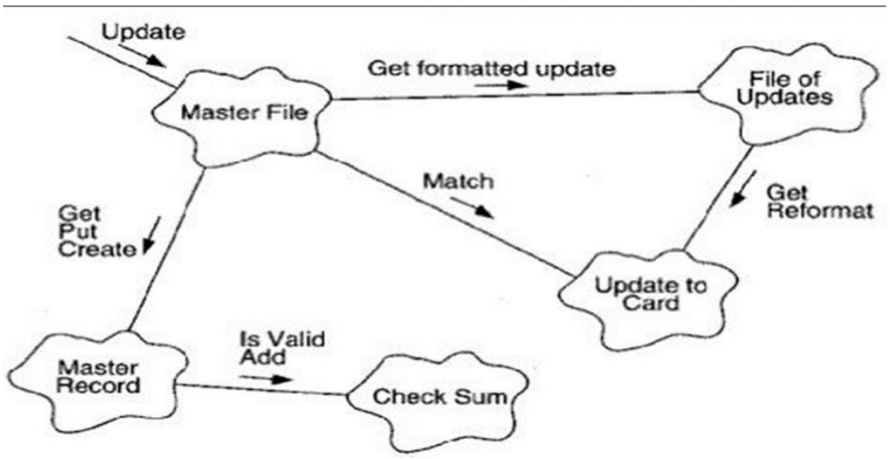
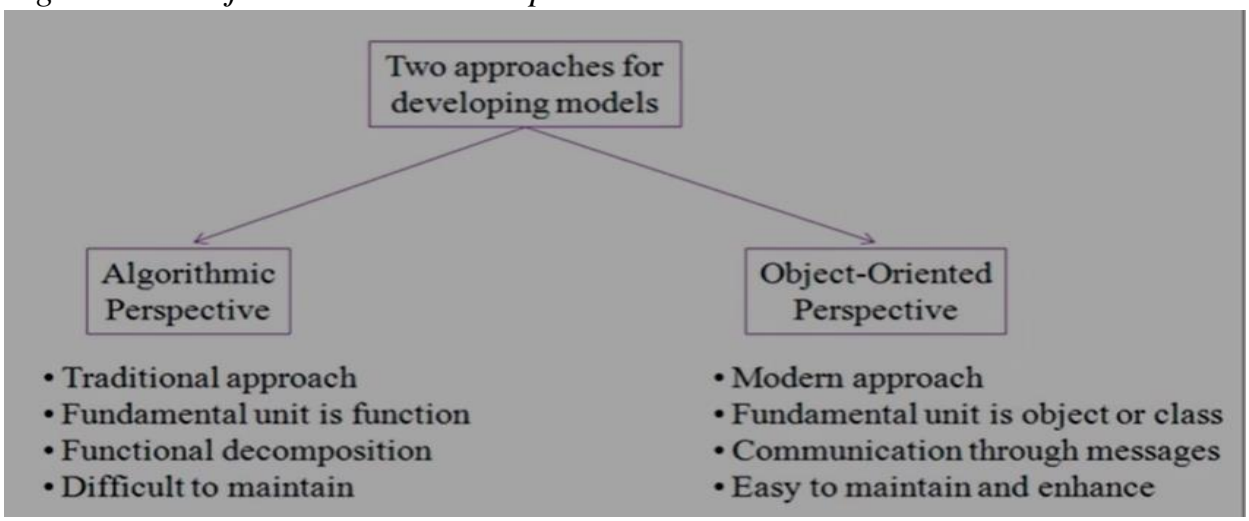


Figure 1.3: Object Oriented decomposition



Algorithmic versus object-oriented decomposition:

1. The algorithmic view highlights the ordering of events and the object-oriented view emphasizes the agents that either cause action or are the subjects upon which these operations act.
2. start decomposing a system either by algorithms or by objects then use the resulting structure as the framework for expressing the other perspective generally object-oriented view is applied because this approach is better at helping us organize the inherent complexity of software systems.
3. object oriented algorithm has a number of advantages over algorithmic decomposition.
4. Object oriented decomposition yields smaller systems through the reuse of common mechanisms, thus providing an important economy of expression and are also more resident to change and thus better able to involve over time and it also reduces risks of building complex software systems.
5. Object oriented decomposition also directly addresses the inherent complexity of software by helping us make intelligent decisions regarding the separation of concerns in a large state space.
6. Process-oriented decompositions divide a complex process, function or task into simpler sub processes until they are simple enough to be dealt with.
7. The solutions of these sub functions then need to be executed in certain sequential or parallel orders in order to obtain a solution to the complex process.
8. Object-oriented decomposition aims at identifying individual autonomous
9. objects that encapsulate both a state and a certain behavior.
10. Then communication among these objects leads to the desired solutions.
11. Although both solutions help dealing with complexity, reasons to believe that an object-oriented decomposition is favorable because, the object-oriented approach provides for a semantically richer framework that leads to decompositions that are more closely related to entities from the real world.
12. Moreover, the identification of abstractions supports (more abstract) solutions to be reused and the object-oriented approach supports the evolution of systems better as those concepts that are more likely to change can be hidden within the objects.

6.On Designing Complex Systems:

1. Engineering as a Science and an Art
2. The meaning of Design
3. The Importance of Model Building
4. The Elements of Software design Methods
5. The models of Object-Oriented Development

Engineering as a Science and an Art:

1. Every engineering discipline involves elements of both science and art.
2. The programming challenge is a large-scale exercise in applied abstraction and thus requires the abilities of the formal mathematician blended with the attribute of the competent engineer.

3. The role of the engineer as artist is particularly challenging when the task is to design an entirely new system.

2.The meaning of Design:

1. In every engineering discipline, design encompasses the discipline approach we use to invent a solution for some problem, thus providing a path from requirements to implementation.
2. The purpose of design is to construct a system that.
    a. Satisfies a given (perhaps) informal functional specification
    b. Conforms to limitations of the target medium
    c. Meets implicit or explicit requirements on performance and resource usage
    d. Satisfies implicit or explicit design criteria on the form of the artifact
    e. Satisfies restrictions on the design process itself, such as its length or cost, or the available for doing the design.
3. According to Stroustrup, the purpose of design is to create a clean and relatively simple internal structure, sometimes also called as architecture.
4. A design is the end product of the design process.

3.The Importance of Model Building:

1. The buildings of models have a broad acceptance among all engineering disciplines largely because model building appeals to the principles of decomposition, abstraction and hierarchy. Each model within a design describes a specific aspect of the system under consideration. Models give us the opportunity to fail under controlled conditions.
2. Evaluate each model under both expected and unusual situations and then after them when they fail to behave as we expect or desire.
3. More than one kind of model is used on order to express all the subtleties of a complex system.

4.The Elements of Software design Methods: Design of complex software system involves an incremental and iterative process. Each method includes the following:

1. **Notation:** The language for expressing each model.
2. **Process:** The activities leading to the orderly construction of the system's mode.
3. **Tools:** The artifacts that eliminate the medium of model building and enforce rules about the models themselves, so that errors and inconsistencies can be exposed.

5.The models of Object-Oriented Development:

1. The models of object-oriented analysis and design reflect the importance of explicitly capturing both the class and object hierarchies of the system under design.
2. These models also over the spectrum of the important design decisions that consider in developing a complex system and so encourage us to craft implementations that embody the five attributes of well-formed complex systems.

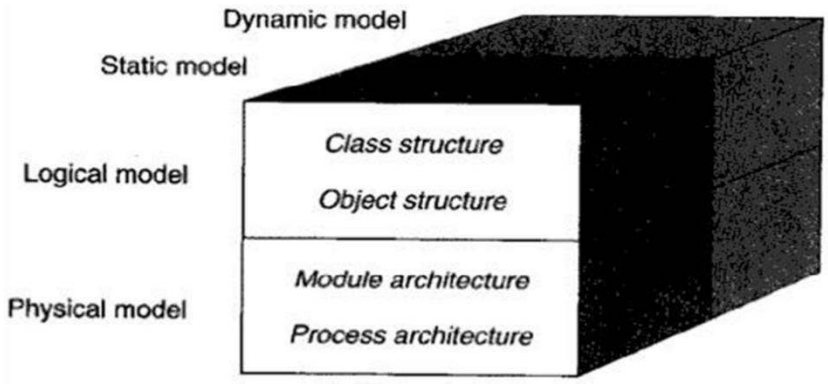


Figure: Models of object-oriented development

- Booch presents a model of object-oriented development that identifies several relevant perspectives. The classes and objects that form the system are identified in a logical model.
- For this logical model, again two different perspectives have to be considered.
- A static perspective identifies the structure of classes and objects, their properties and the relationships classes and objects participate in.
- A dynamic model identifies the dynamic behavior of classes and objects, the different valid states they can be in and the transitions between these states.
- Besides the logical model, also a physical model needs to be identified.
- This is usually done later in the system's lifecycle. The module architecture identifies how classes are kept in separately compliable modules and the process architecture identifies how objects are distributed at run-time over different operating system processes and identifies the relationships between those.
- Again, for this physical model a static perspective is defined that considers the structure of module and process architecture and a dynamic perspective identifies process and object activation strategies and inter- process communication.
- Object-orientation has not, however, emerged fully formed.
- In fact, it has developed over a long period, and continues to change.

Case Study: System Architecture: Satellite-Based Navigation

1.Inception

1.1.Requirements for the Satellite Navigation System

1.2. Defining the Boundaries of the Problem

1.3. Determining Mission Use Cases

1.4. Determining System Use Cases

2. Elaboration

2.1. Developing a Good Architecture

2.2 Defining Architectural Development Activities

2.3 Validating the Proposed System Architecture

2.4 Allocating Nonfunctional Requirements and Specifying Interfaces

2.5 Stipulating the System Architecture and Its Deployment

2.6 Decomposing the System Architecture

3. Construction

4. Post -Transition

Satellite Based Navigation

- 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.

- 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

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" .

•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".

•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.

1.1.Requirements for the Satellite Navigation System

1.2. Defining the Boundaries of the Problem

1.3. Determining Mission Use Cases

1.4. Determining System Use Cases

1.1. 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

•**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

1.2.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 5 –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 environment.

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

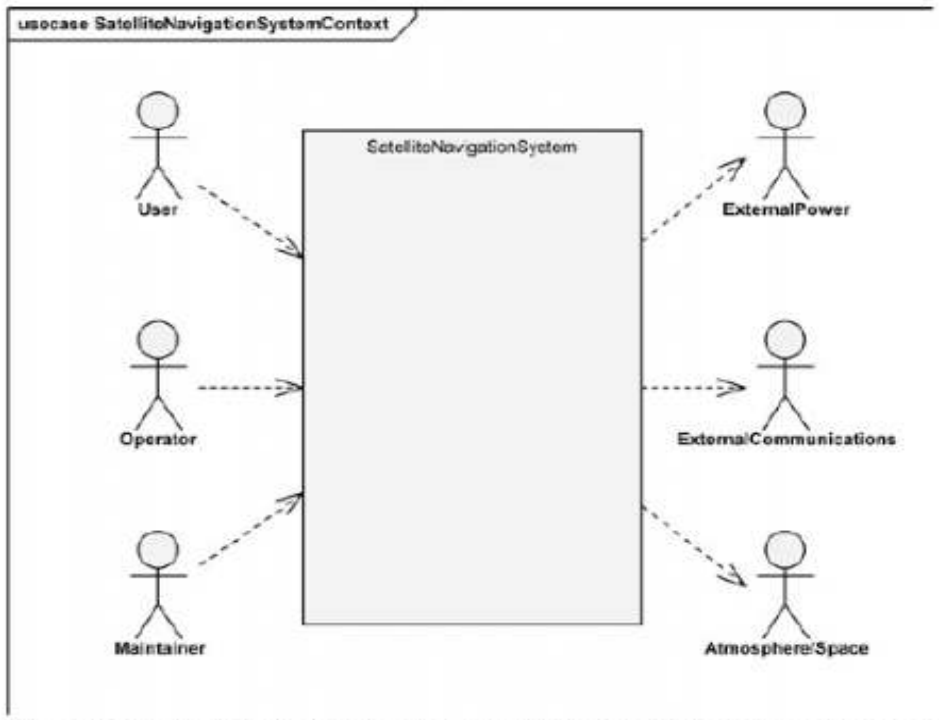


Figure 5-1 The Satellite Navigation System Context Diagram

•In addition to the functional requirements, we've been given high-level non-functional 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.

•After reviewing both the vision and the requirements, (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 fulfil their missions.

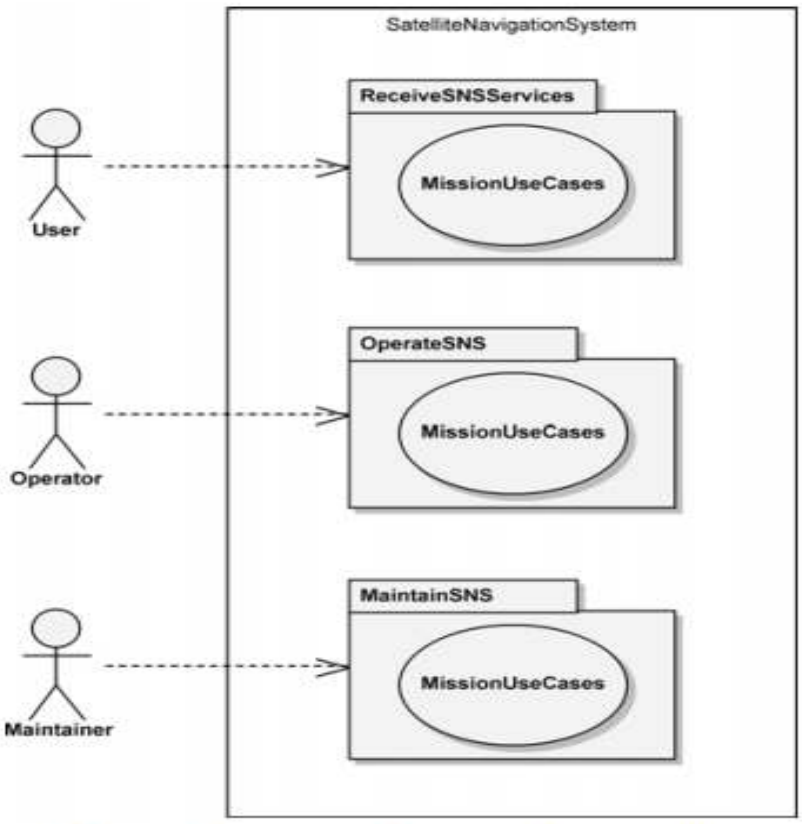


Figure 8–2 Packages for the SNS Mission Use Cases

1.3. 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).

• Begin by developing the mission use case 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 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, the highest-level logical architecture consists of four segments: Ground, Launch, Satellite, and User.

•A conceptual architecture at the level of a package diagram like the one shown in Figure 5 –3, we can begin our analysis by working with domain experts to articulate the primary mission use cases that detail the system's desired behaviour.

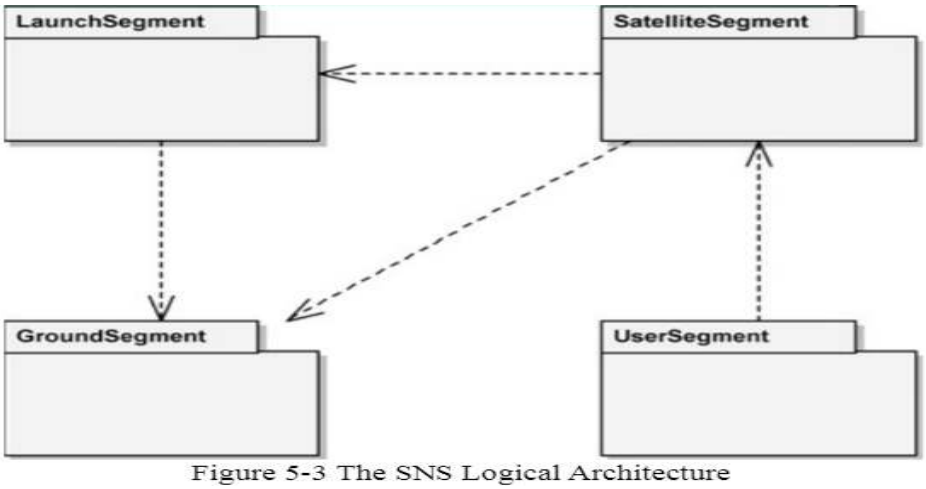


Figure 5-3 The SNS Logical Architecture

• "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.

•Typically, though, our analysis employs activity diagram modelling 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 Operate SNS mission use case package.

•For the remainder 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.

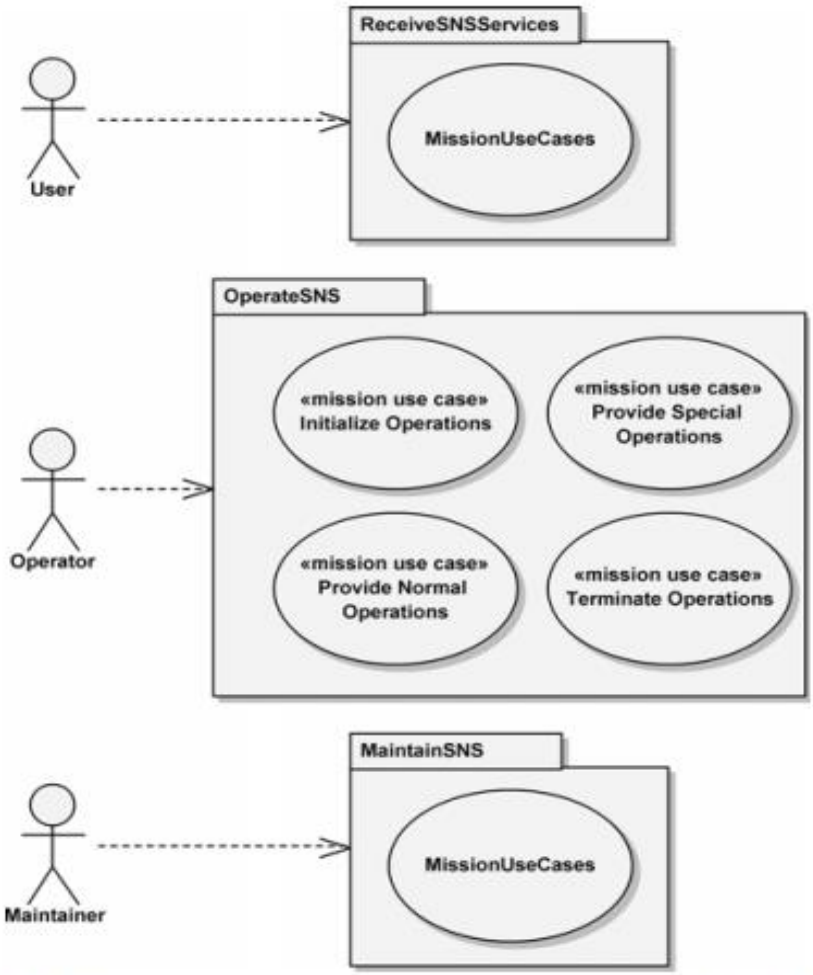


Figure 8–4 Refining the `OperateSNS` Mission Use Case Package

1.4. Determining System Use Cases

• 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. 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.

• 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.

•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.

•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.

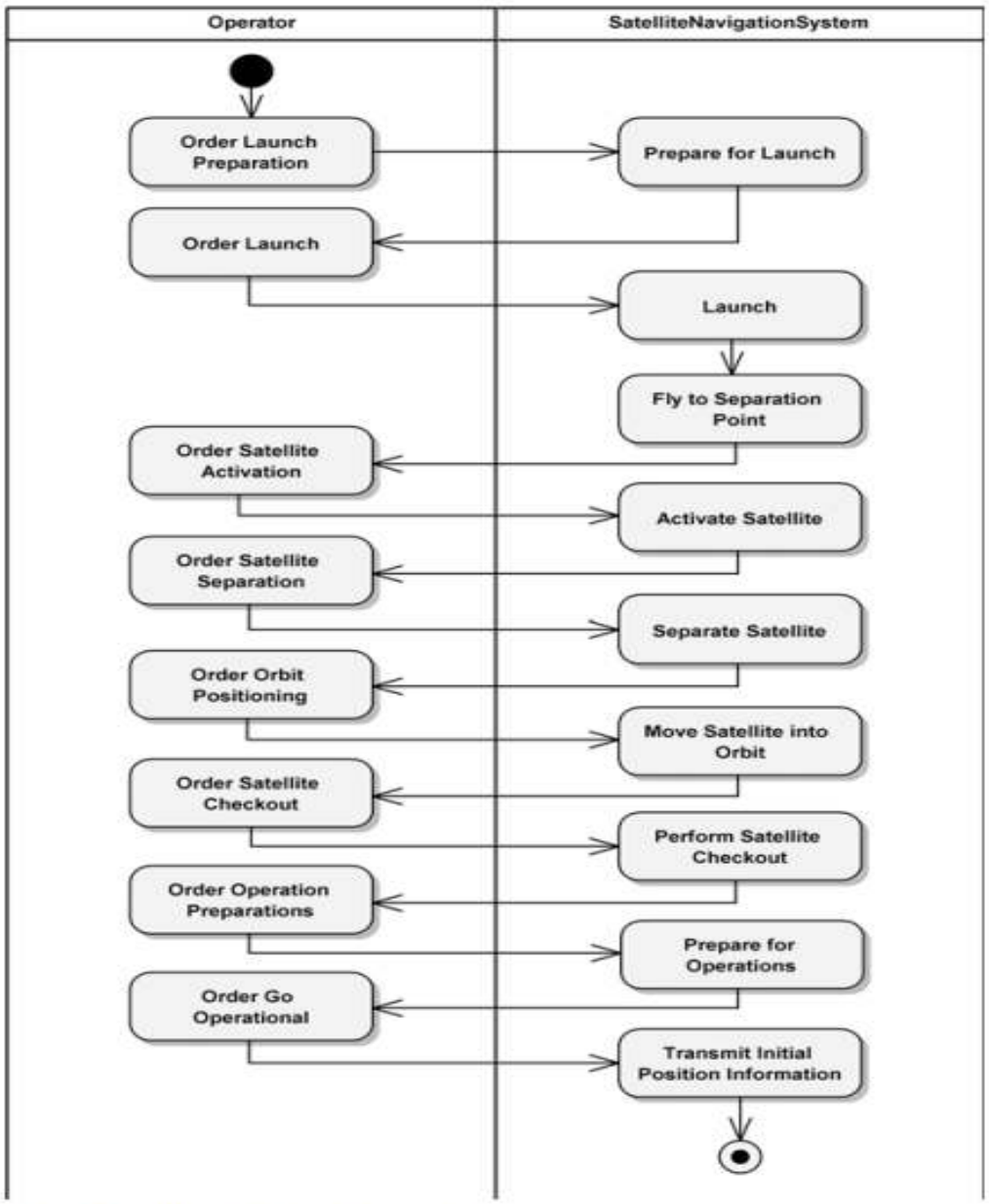


Figure 8–5 The Black-Box Activity Diagram for `Initialize Operations`

Figure 5-5 The Black-Box Activity Diagram for Initialize Operations

•Figure 5 –6 shows the updated use case diagram. used the Initialize Operations 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».

• find this modeling approach to be useful and clear; however, each development team needs to determine and document its chosen techniques.

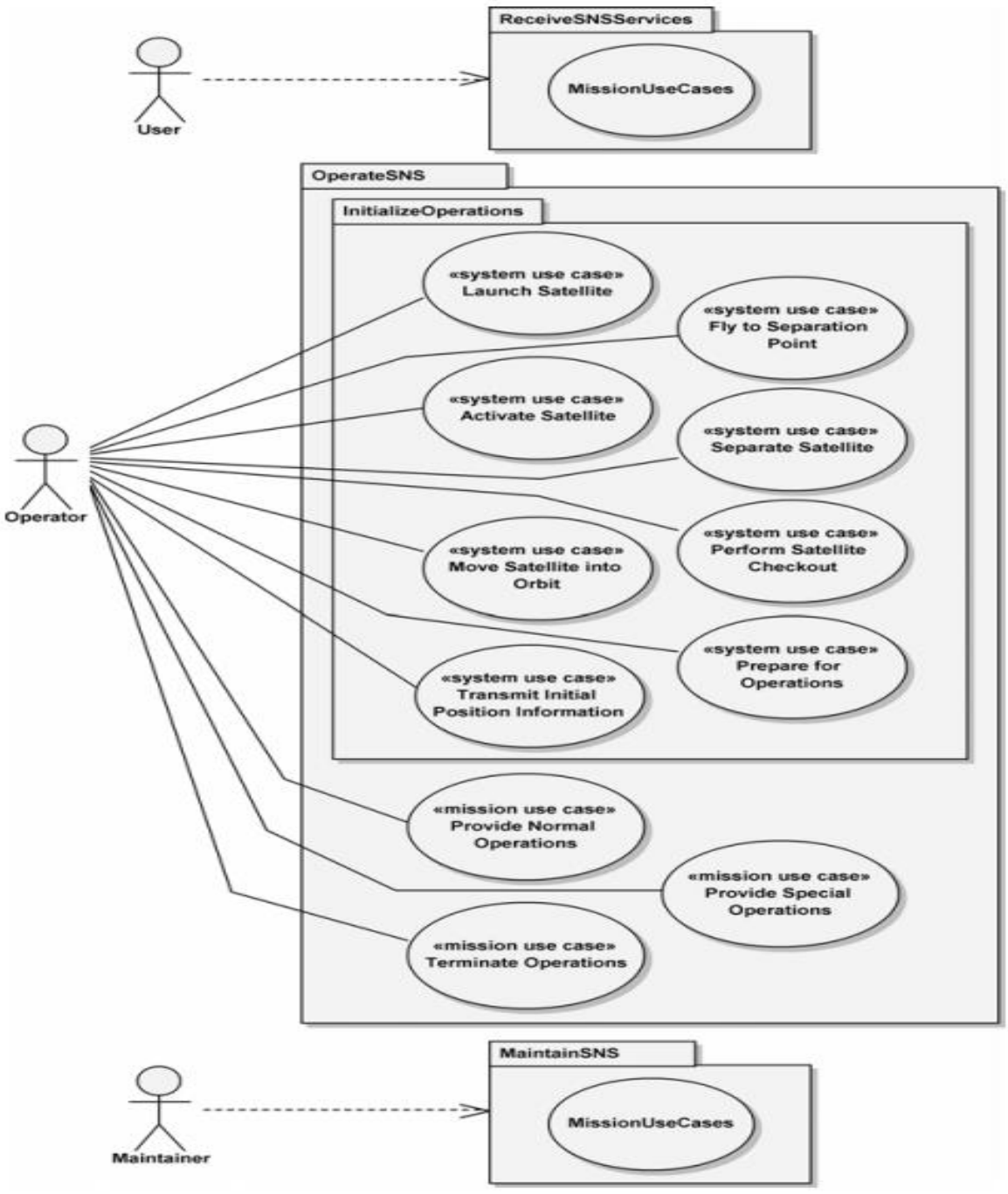


Figure 8–6 System Use Cases for `Initialize Operations`

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.

2. Elaboration

2.1.Developing a Good Architecture

2.2 Defining Architectural Development Activities

2.3 Validating the Proposed System Architecture

2.4 Allocating Nonfunctional Requirements and Specifying Interfaces

2.5 Stipulating the System Architecture and Its Deployment

2.6 Decomposing the System Architecture

2.1.Developing a Good Architecture

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. 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.

2.2 Defining Architectural Development Activities

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.

2.3 Validating the Proposed System Architecture

Beginning with the black-box activity diagram for Initialize Operations presented earlier in Figure 5–5, we allocate the system functionality, shown in the Satellite Navigation System partition, to one or more of its constituent segments: Ground, Launch, Satellite, or User.

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 behaviour. 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

The white-box activity diagram for Initialize Operations presents the results of analyzing only a portion of the functionality contained within the Operate SNS 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

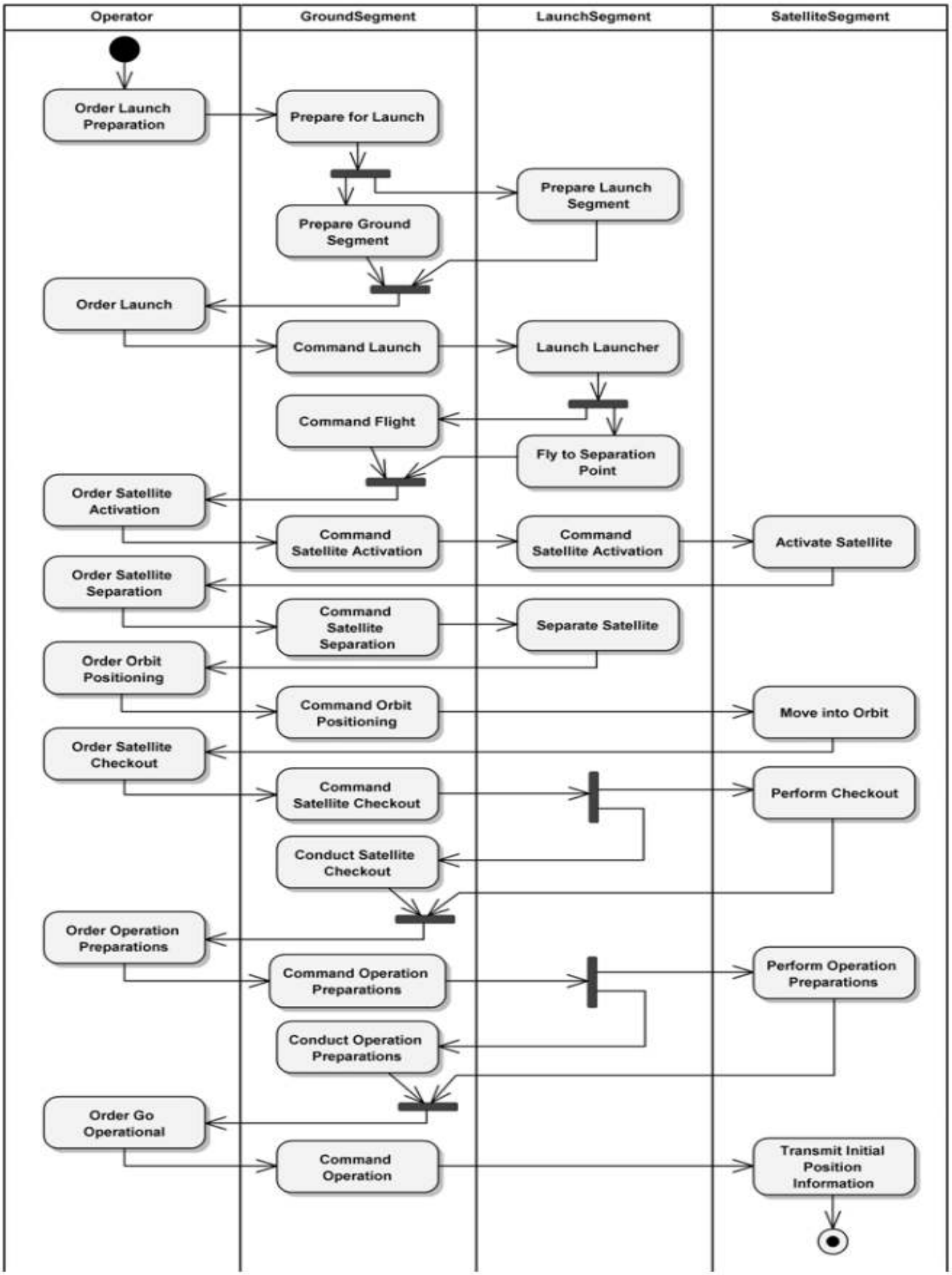


Figure 8–7 The White-Box Activity Diagram for `Initialize Operations`

2.4 Allocating Nonfunctional Requirements and Specifying Interfaces

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.

potential requirements alluded to by the design constraints given:

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

A. The constraint "Compatibility with international standards" drove our specification of the external actor Atmosphere/Space, as discussed earlier.
B. Interact with the national and international agencies that regulate the use of the airwaves to determine,
C. 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.
D. This means that the Ground Segment, Launch Segment (at least during the flight phase), and Satellite Segment now must fulfil the external interface responsibilities of the Satellite Navigation System.
E. 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

2.5 Stipulating the System Architecture and Its Deployment

1. The component diagram may be used to hierarchically decompose a system, and thus it here
2. represents the highest-level abstraction of the Satellite Navigation System architecture, that is, its
3. segments and their relationships.
4. Figure 8–8 illustrates two ways to represent the interface between segments, the ball-and-socket notation (Launch Support interface) and the dashed dependency line connecting the required and provided interfaces (Position Information interfaces).
5. Looking back at Figure 5–1, we see that three of the system actors are not accounted for by the SNS interfaces shown in Figure5–8: External Power, External Communications, and Atmosphere/Space.
6. These actors provide important services to the Satellite Navigation System; however, they are not central to our focus on developing its logical architecture

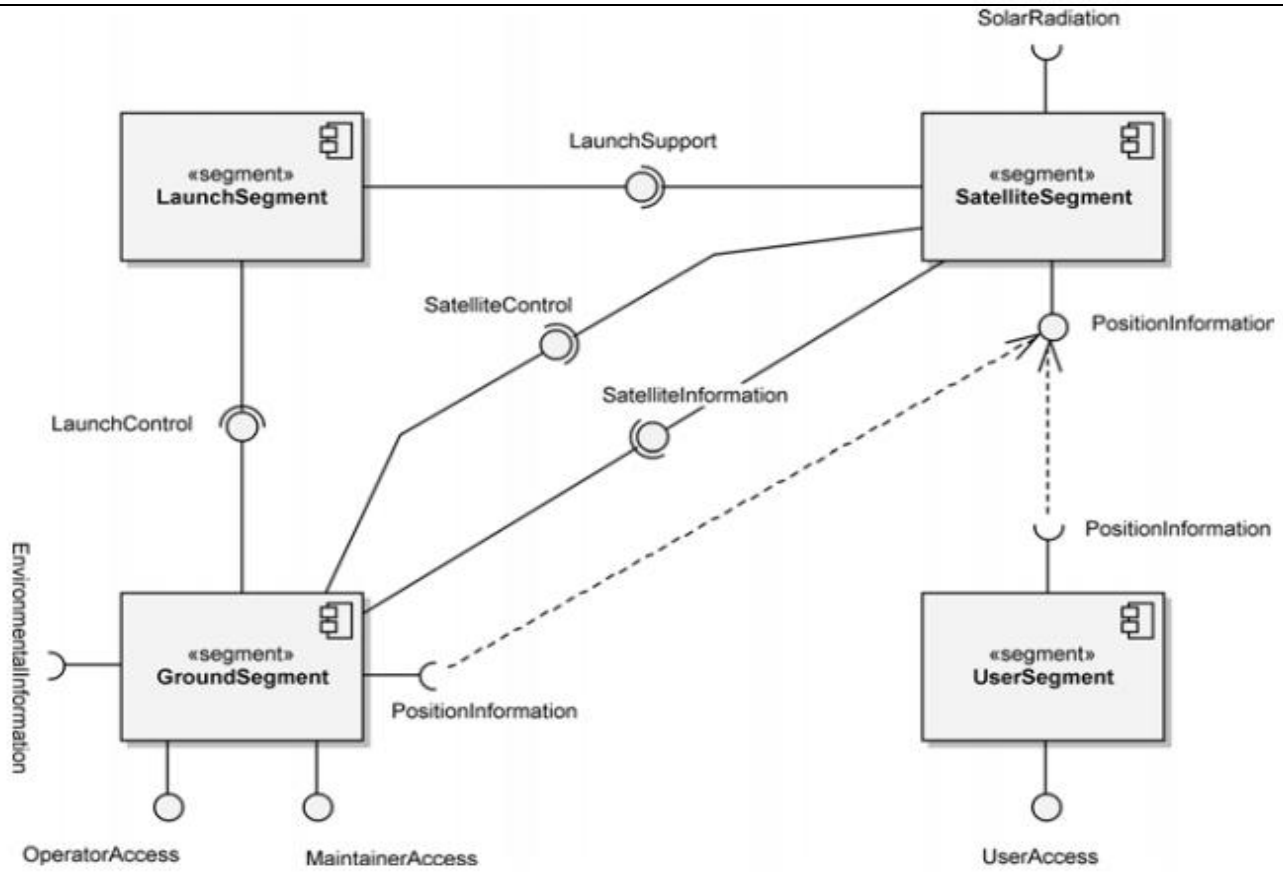


Figure 8–8 The Component Diagram for the Satellite Navigation System

Figure 8 –8 shows the deployment of the components represented.recognize that this is not a typical use of the notation; typically, 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 .

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

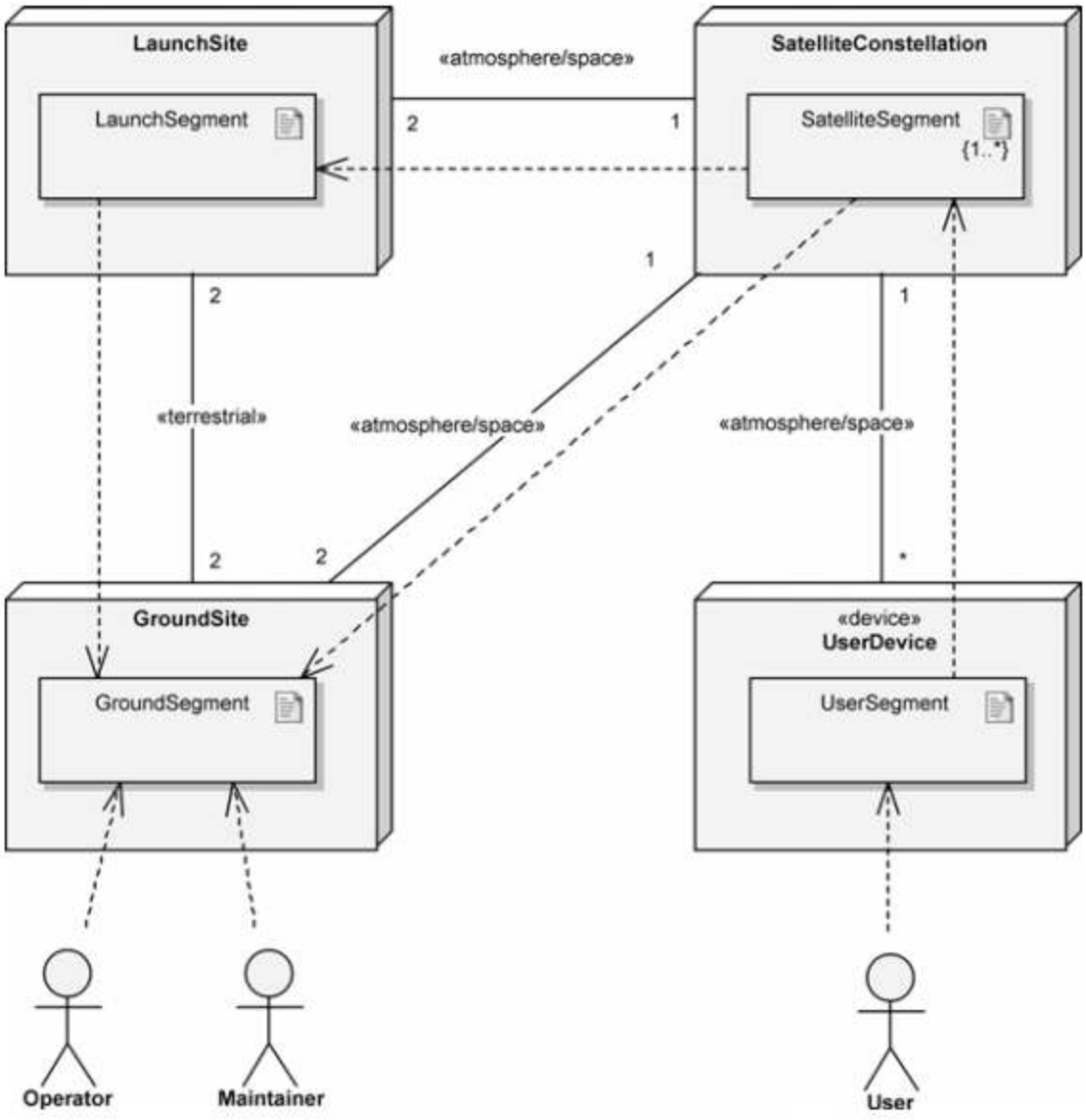


Figure 8–9 The Deployment of SNS Segments

2.6 Decomposing the System Architecture

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.

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.

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. 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 Ground Segment use cases from these allocated system actions
- Performed black-box analysis for the Ground Segment's Control Launch use case to determine its actions
- Performed white-box analysis of the Ground Segment's Control Launch actions to allocate them across its subsystems

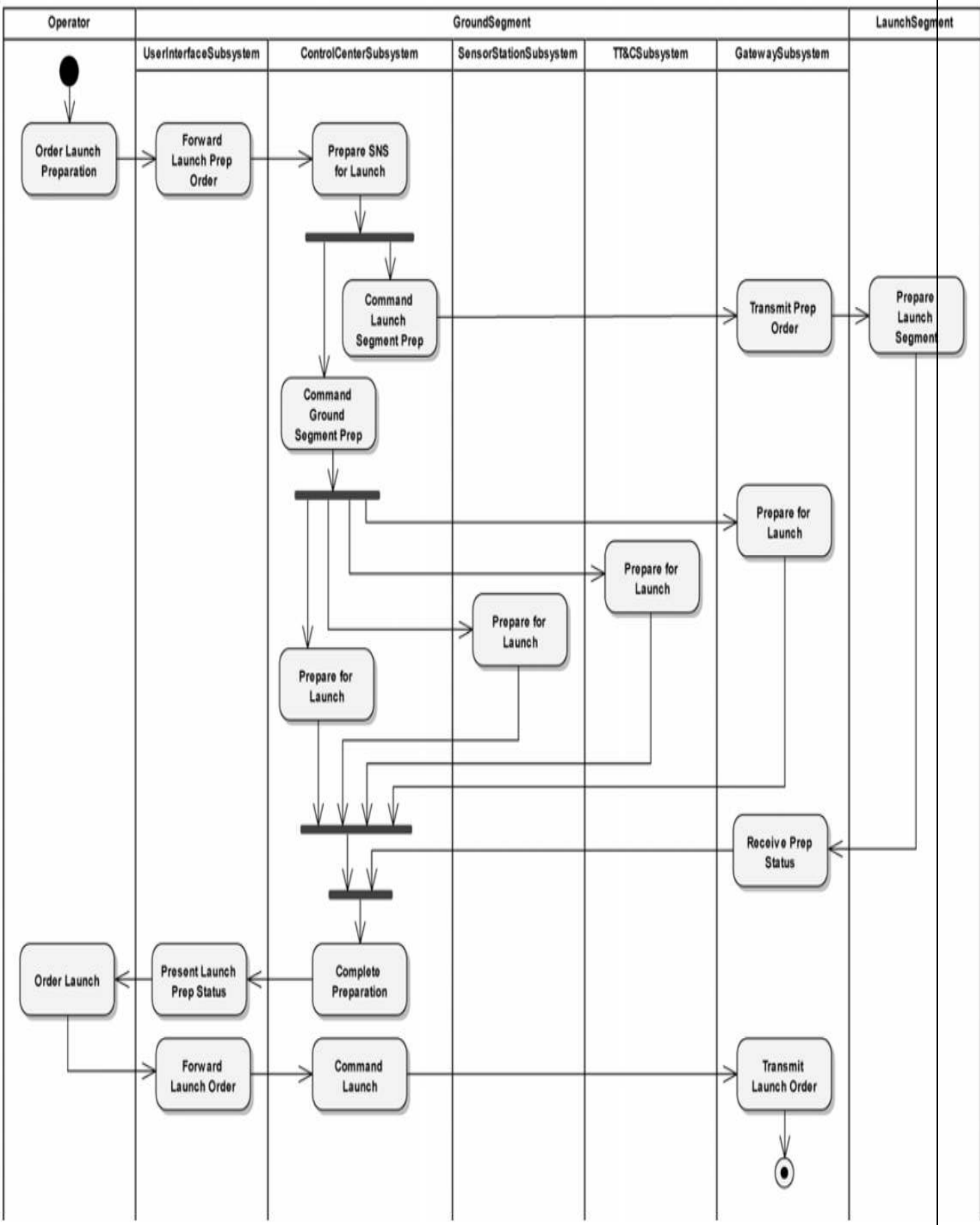


Figure 8–10 The White-Box Activity Diagram for `Control Launch`

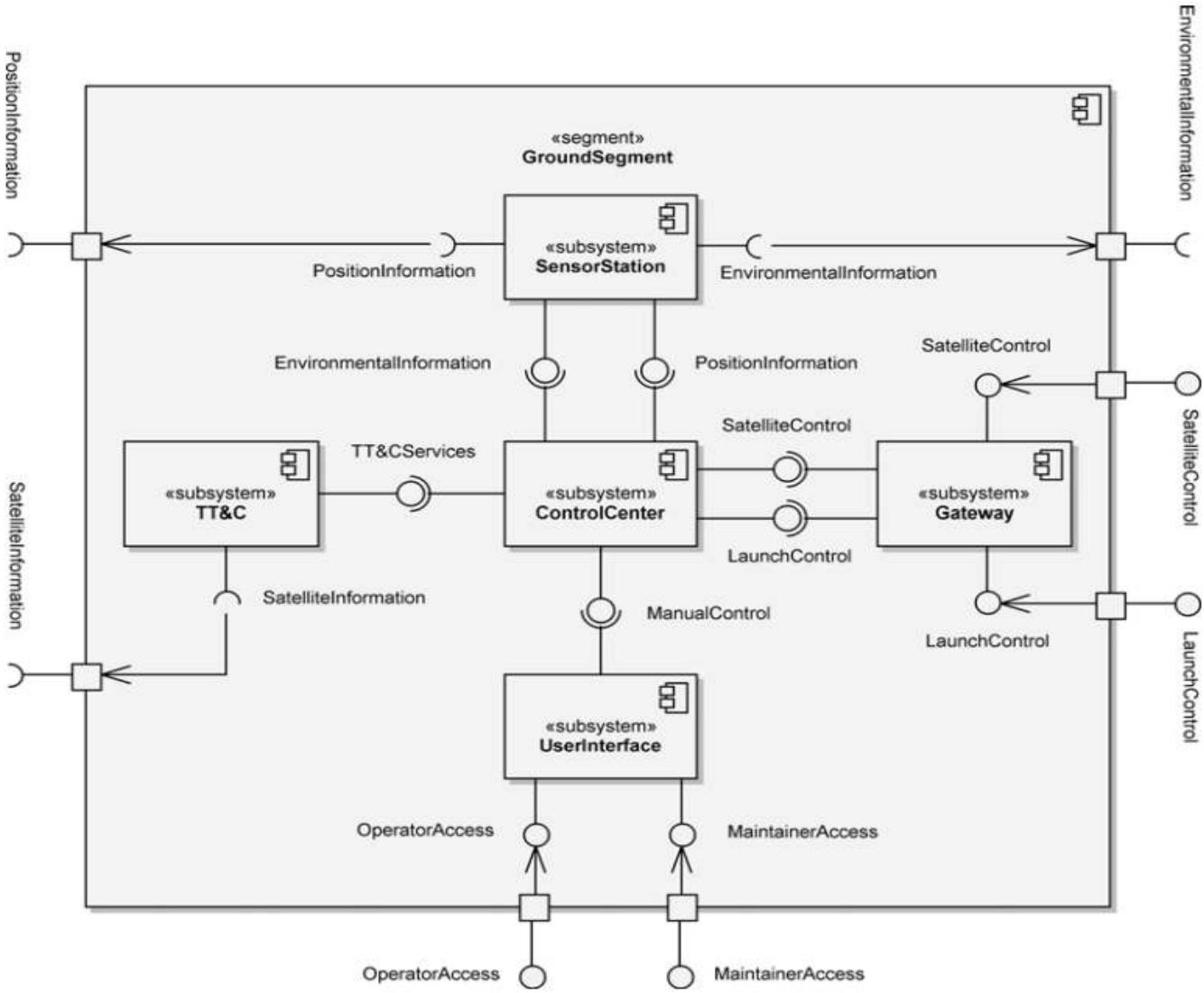


Figure 8–11 The Logical Architecture of the `GroundSegment`

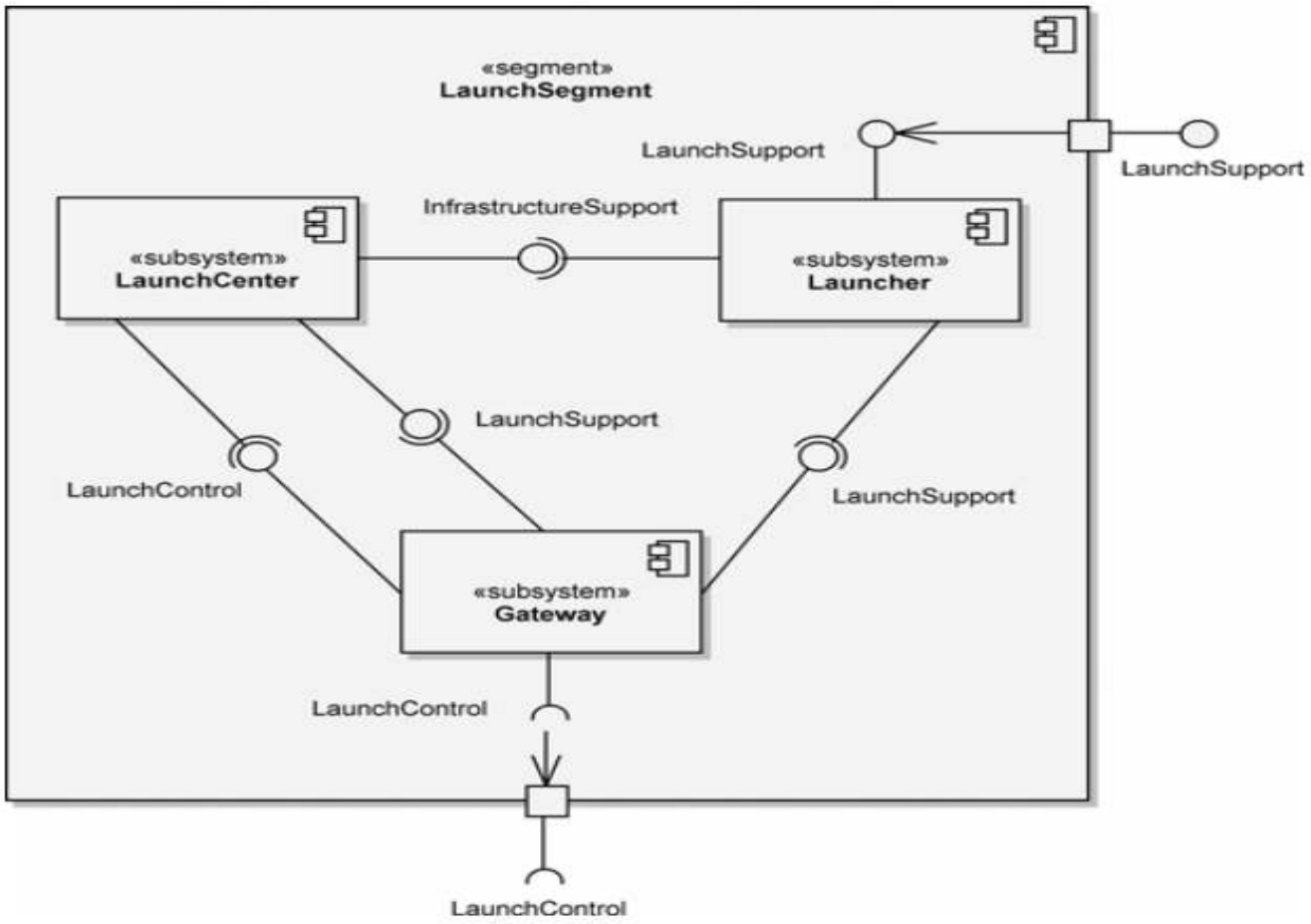


Figure 8–12 The Logical Architecture of the `LaunchSegment`

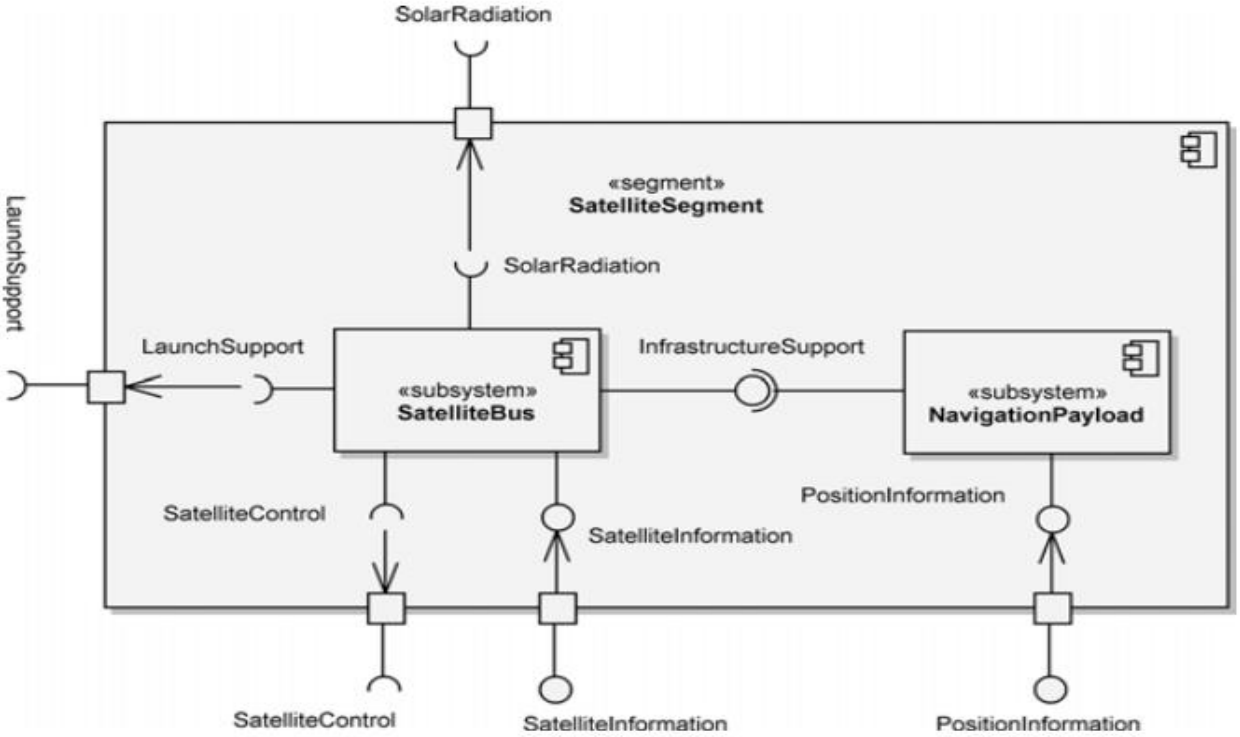


Figure 8–13 The Logical Architecture of the `SatelliteSegment`

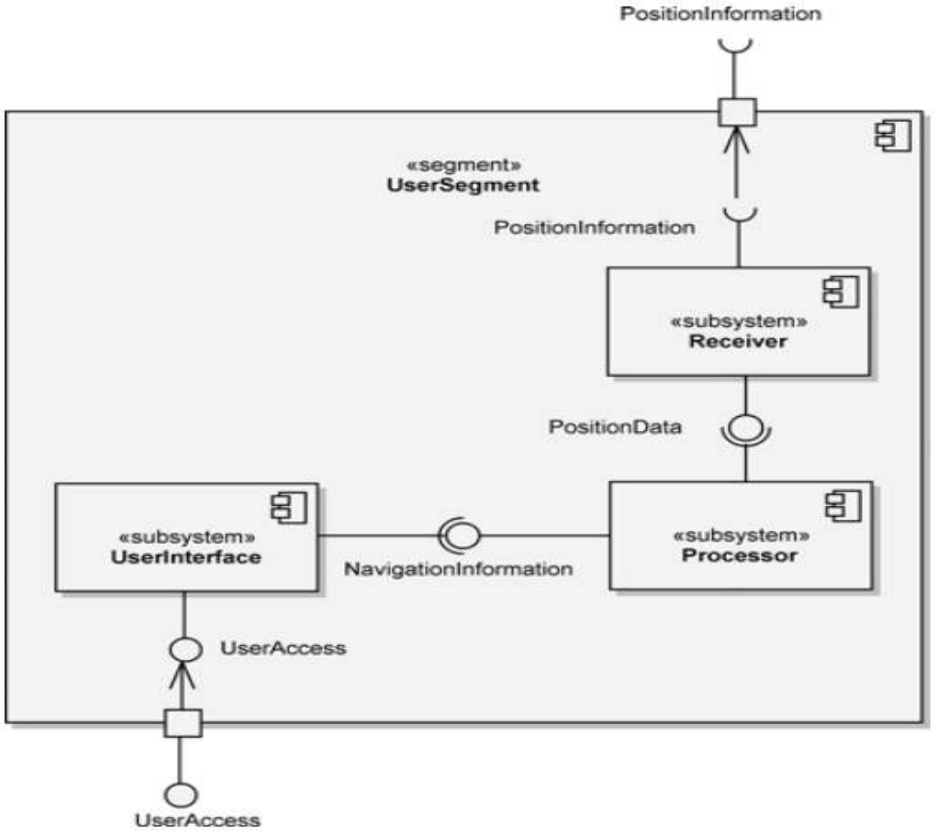


Figure 8–14 The Logical Architecture of the `UserSegment`

1.3Construction

2. At the end of the Elaboration phase, a stable architecture of our system should have been developed.
3. 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.
4. 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

1.4 Post - Transition

1. The Satellite Navigation System's original nonfunctional requirements included two that caused us to develop a flexible architecture: extensibility and long service life.
2. This long service life dictates, in addition to many other aspects, a design that is extensible to ensure the reliable provision of desired functionality.
3. 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.
4. Investigate how well our SNS design has met these requirements as we add new functionality and also change the system's target hardware