UNIT-1

Object:

An object is a real-world element in an object–oriented environment that may have a physical or a conceptual existence. Objects can be modelled according to the needs of the application. An object may have a physical existence, like a customer, a car, etc.; or an intangible conceptual existence, like a project, a process, etc.

Object Oriented Analysis and Design (OOAD):

Object-oriented analysis and design (OOAD) is a popular technical approach for analyzing and designing an application, system, or business by applying object-oriented paradigm, as well as using visual modeling throughout the development life cycle for better communication and product quality.

Object Oriented Analysis (OOA):

The main difference between Object-Oriented Analysis and other forms of analysis is that by the object-oriented approach we organize requirements around objects, which integrate both behaviors (processes) and states (data) modeled after real world objects that the system interacts with.

The primary tasks in object-oriented analysis (OOA) are:

- Find the objects
- Organize the objects
- Describe how the objects interact
- Define the behavior of the objects
- Define the internals of the objects

Object Oriented Design (OOD):

Object Oriented Design involves implementation of the conceptual model produced during Object Oriented Analysis. OOD concepts in the analysis model which are technology independent are mapped onto implementing classes, Constraints are identified and Interfaces are designed resulting in a model for the solution domain.

Object Oriented Paradigm:

Object oriented paradigm based upon objects (having data in the form of fields often known as attributes and code in the form of procedures often known as methods) that aims to incorporate the advantages of modularity and reusability.

1.1 The Structure of Complex Systems:

Systems with a set of parts are called as elements and a set of connections between these parts are called relations. These parts can be ordered or unordered.

Downloaded by TONY 562 (tony2802b@gmail.com)

E.g.: Parts of a car, Organs in our body.

Software might be referred to as simple or complex depending upon their functionality and behavior. Generally, industrial-strength software are more complex than those developed individually or by user developers.

Time and space are considered to be general complexities. Additionally, maintaining integrity of hundreds of thousands of records while allowing concurrent updates and queries; or managing command and control of real-world entities like air traffic are also examples of complex systems.

Structure of a Personal Computer:

A device of moderate complexity, which has been evolved from small working sub systems like gate architecture, for instance.

- Composed of the same major elements – CPU, monitor, keyboard, secondary storage
- An element can be taken and decomposed further and can be studied separately.
- The collaborative activity of each major part (hierarchically arranged) causes a computer system to function.
- The hierarchy of the computer system also represents different levels of abstraction – each built upon another and in each level of abstraction, a collection of devices are found to be collaborating one another to provide functions to higher layers.

Structure of a Plants and Animals:

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.

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.

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.

1.2 The inherently Complexity of Software:

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

Downloaded by TONY 562 (tony2802b@gmail.com)

a. The complexity of the problem domain

- ☐ Complex requirements
- ☐ Decay of system

Often, although software developers have the knowledge and skills required to develop software they usually lack detailed knowledge of the application domain of systems. 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.

External complexity usually springs from the difference of thoughts that exists between the users of a system and its developers. Users may have only vague ideas of what they want in a software system.

Users and developers have different perspectives on the nature of the problem and make different assumptions regarding the nature of the system. A further complication is that the requirement of a software system is often change during its development.

It is software maintenance when we correct errors, evolution when we respond to changing environments and preservations, when we continue to use extraordinary means to keep an ancient and decaying piece of software in operation.

b. The Difficulty of Managing the Development Process

- Management problems
- Need of simplicity

The second reason is the complexity of the software development process. Complex software intensive systems cannot be developed by single individuals. They require teams of developers.

Developer must strive to write the less code by using the clever and powerful techniques. Even if we decompose our implementation in meaningful ways, we still end up with hundreds and sometimes even thousand modules.

No single person can totally understand the complexity of system. There are always significant challenges associated with team development more developers' means more complex communication and hence more difficult coordination.

c. The flexibility possible through software

- ☐ Software is flexible and expressive and thus encourages highly demanding requirements, which in turn lead to complex implementations which are difficult to assess.

The third reason is the danger of flexibility. Flexibility leads to an attitude where developers develop system components themselves rather than purchasing them from somewhere else. The software development is different: most of the software companies develop every single component from scratch. Construction industry has standards for quality of row materials, few such standards exist in the software industry.

Downloaded by TONY 562 (tony2802b@gmail.com)

d. The problem of characterizing the behavior of discrete systems

- Numerous possible states
- Difficult to express all states

The final reason for complexity according to Booch is related to the difficulty in describing the behavior of software systems. 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. This is because to describe behavior, it is not sufficient to list the properties of the system. It is also necessary to describe the sequence of the values that these properties take over time.

Within a large application, there may be hundreds or even thousands of variables well as more than one thread of control. The entire collection of these variables as well as their current values and the current address within the system constitute the present state of the system with discrete states. Discrete systems by their very nature have a finite numbers of possible states.

e. The Consequences of Unrestrained Complexity

The more complex the system, the more open it is to total breakdown. Rarely would a builder think about adding a new sub-basement to an existing 100-story building; to do so would be very costly and would undoubtedly invite failure. Amazingly, users of software systems rarely think twice about asking for equivalent changes. Besides, they argue, it is only a simple matter of programming.

1.3 Attributes of a complex system:

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

1. Hierarchical and interacting subsystems

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. Arbitrary determination of primitive components

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

Intra-component linkages are generally stronger than inter-component linkages. 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

Downloaded by TONY 562 (tony2802b@gmail.com)

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

Booch has identified five properties that architectures of complex software systems have in common. Firstly, every complex system is decomposed into a hierarchy of subsystems. This decomposition is essential in order to keep the complexity of the overall system manageable. These subsystems, however, are not isolated from each other, but interact with each other.

Very often subsystems are decomposed again into subsystems, which are decomposed and so on. The way how this decomposition is done and when it is stopped, i.e. which components are considered primitive, is rather arbitrary and subject to the architects decision.

The decomposition should be chosen, such that most of the coupling is between components that lie in the same subsystem and only a loose coupling exists between components of different subsystem. This is partly motivated by the fact that often different individuals are in charge with the creation and maintenance of subsystems and every additional link to other subsystems does imply an higher communication and coordination overhead.

Certain design patterns re-appear in every single subsystem. Examples are patterns for iterating over collections of elements, or patterns for the creation of object instances and the like.

The development of the complete system should be done in slices so that there is an increasing number of subsystems that work together. This facilitates the provision of feedback about the overall architecture.

1.4 Organized and Disorganized Complexity

Simplifying Complex Systems

- Usefulness of abstractions common to similar activities e.g. driving different kinds of motor vehicle
- Multiple orthogonal hierarchies e.g. structure and control system
- Prominent hierarchies in object-orientation " class structure " " object structure " e. g. engine types, engine in a specific car.

One mechanism to simplify concerns in order to make them more manageable is to identify and understand abstractions common to similar objects or activities. We can use a car as an example (which are considerable complex systems). Understanding common abstractions in this

Downloaded by TONY 562 (tony2802b@gmail.com)

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).

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 is the simplest form of related hierarchy. It forms a canonical representation for object oriented analysis.

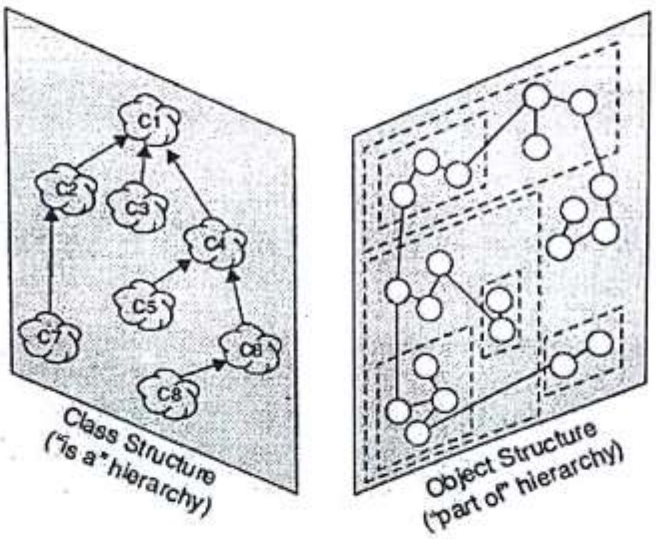

C1
C2
C3
C4
C5
C6
C7
C8
Class Structure ("is a" hierarchy)
Object Structure ("part of" hierarchy)

Fig 1.4.1 The Key Hierarchies of Complex

Systems **The canonical form of a complex system:**

The discovery of common abstractions and mechanisms greatly facilitates are standing of complex system. 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. For example an aircraft may be studied by decomposing it into its propulsion system. Flight control system and so on the decomposition represent a structural or "part of" hierarchy. The complex system also includes an "Is A" hierarchy. 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 system takes on the same (canonical) form as shown in figure. There are two orthogonal hierarchies of system, its class structure and the object structure.

Downloaded by TONY 562 (tony2802b@gmail.com)

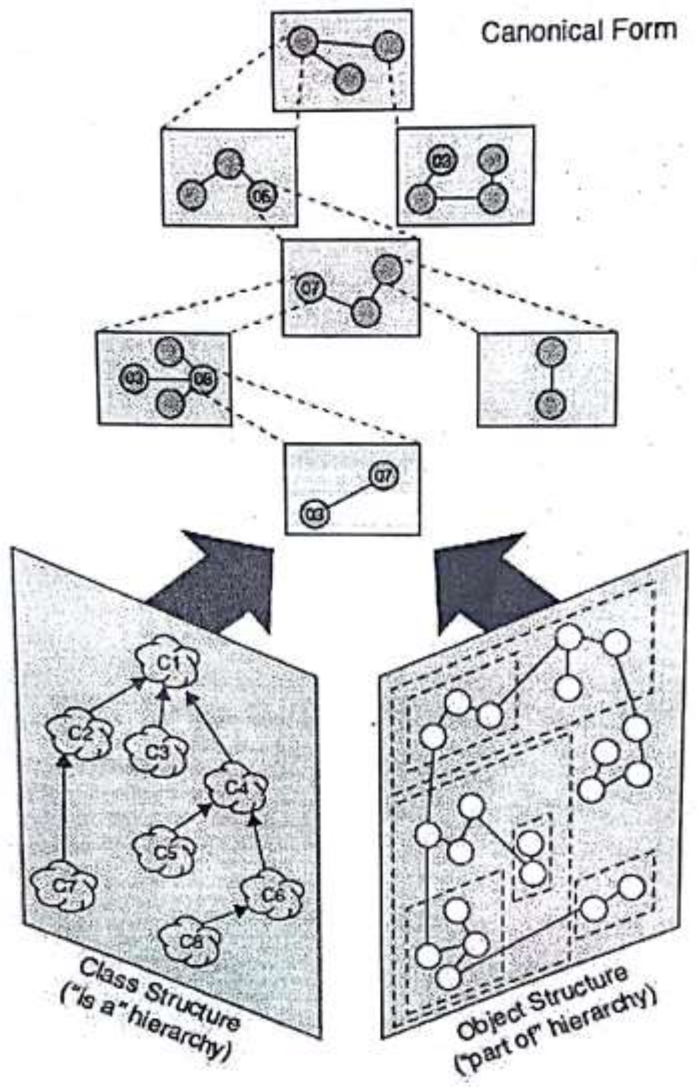

Canonical Form
Class Structure ("is a" hierarchy)
Object Structure ("part of" hierarchy)
C1
C2
C3
C4
C5
C6
C7
C8

Figure 1.4: Canonical form of a complex system

The figure 1.4 represents the relationship between two different hierarchies: 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. 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. The object structure defines the 'part- of' representation. 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. The two hierarchies are not entirely orthogonal as objects are instances of certain classes. The relationship between these two hierarchies is shown by identifying the instance-of relationship as well. 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. The point in identifying classes is therefore to have a vehicle to describe only once all properties that all instances of the class have.

Approaching a Solution

Downloaded by TONY 562 (tony2802b@gmail.com)

The Limitations of the human capacity for dealing with complexity:

- Dealing with complexities
- Memory
- Communications

When we devise a methodology for the analysis and design of complex systems, we need to bear in mind the limitations of human beings, who will be the main acting agents, especially during early phases. Unlike computers, human beings are rather limited in dealing with complex problems and any method need to bear that in mind and give as much support as possible.

Human beings are able to understand and remember fairly complex diagrams, though linear notations expressing the same concepts are not dealt with so easily. This is why many methods rely on diagramming techniques as a basis. The human mind is also rather limited. Miller revealed in 1956 that humans can only remember 7 plus or minus one item at once. Methods should therefore encourage its users to bear these limitations in mind and not deploy overly complex diagrams.

The analysis process is a communication intensive process where the analyst has to have intensive communications with the stakeholders who hold the domain knowledge. Also the design process is a communication intensive process, since the different agents involved in the design need to agree on decompositions of the system into different hierarchies that are consistent with each other.

1.5 Bringing Order to chaos

Principles that will provide basis for development

1. **The Role of Abstraction:** Abstraction is an exceptionally powerful technique for dealing with complexity. Unable to master the entirely of a complex object, we chose to ignore its inessential details, dealing instead with the generalized, idealized model of the object. For example, when studying about how photosynthesis works in a plant, we 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. In general abstraction assists people's understanding by grouping, generalizing and chunking information.

   Object- orientation attempts to deploy abstraction. 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:** Identifying the hierarchies within a complex software system makes understanding of the system very simple. The object structure is important because it illustrates how different objects collaborate with one another through pattern of interaction (called mechanisms). By classifying objects into groups of related abstractions (for example,

Downloaded by TONY 562 (tony2802b@gmail.com)

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.

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. The 'is-a' hierarchy also identifies concepts, such as attributes or operations, that are common to a number of classes and instances thereof. 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. But note that there are many other forms of patterns which are nonhierarchical: interactions, 'relationships'.

3. **The role of Decomposition:** Decomposition is important techniques for copying with complexity based on the idea of divide and conquer. In dividing a problem into a sub problem the problem becomes less complex and easier to overlook and to deal with. Repeatedly dividing a problem will eventually lead to sub problems that are small enough so that they can be conquered. 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. The history of computing has seen two forms of decomposition, process-oriented (Algorithmic) and object-oriented decomposition.

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

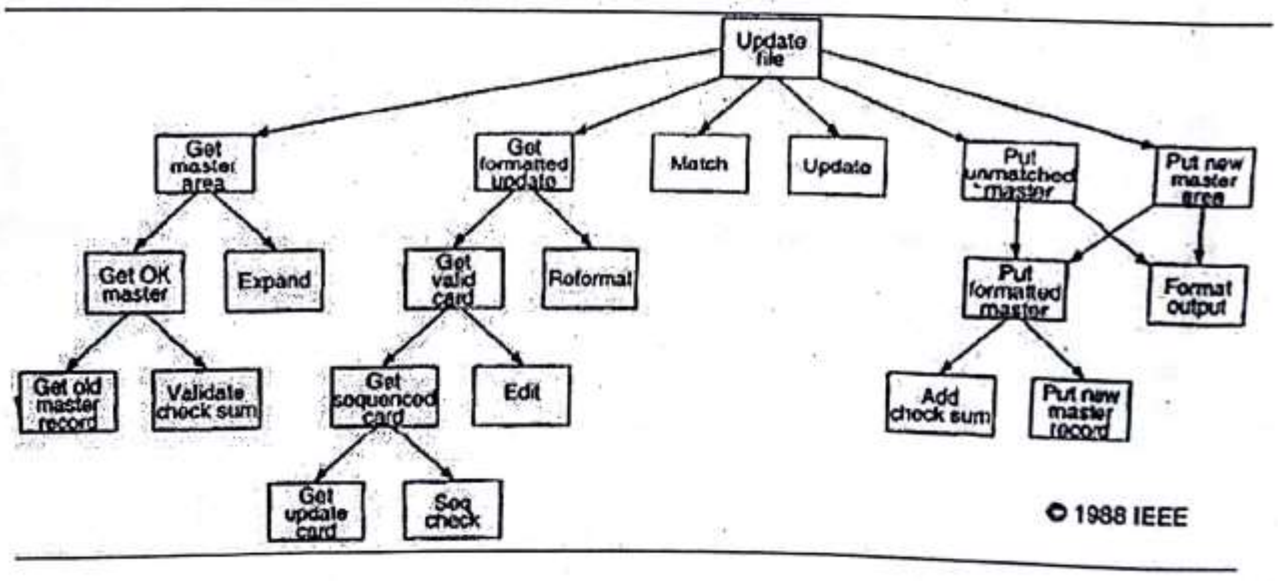


Figure: Algorithmic decomposition

b. **Object oriented decomposition:** Objects are identified as Master file and check sum which derive directly from the vocabulary of the problem as shown in figure. We know the world as a set of autonomous agents that collaborate to perform some higher level behavior. Get formatted update thus does not exist as an independent algorithm; rather it is an operation associated with the object file of updates. Calling this operation creates another object, update to card. In this manner, each object in our solution embodies its own unique behavior .Each

Downloaded by TONY 562 (tony2802b@gmail.com)

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. Here decomposition is based on objects and not algorithms.

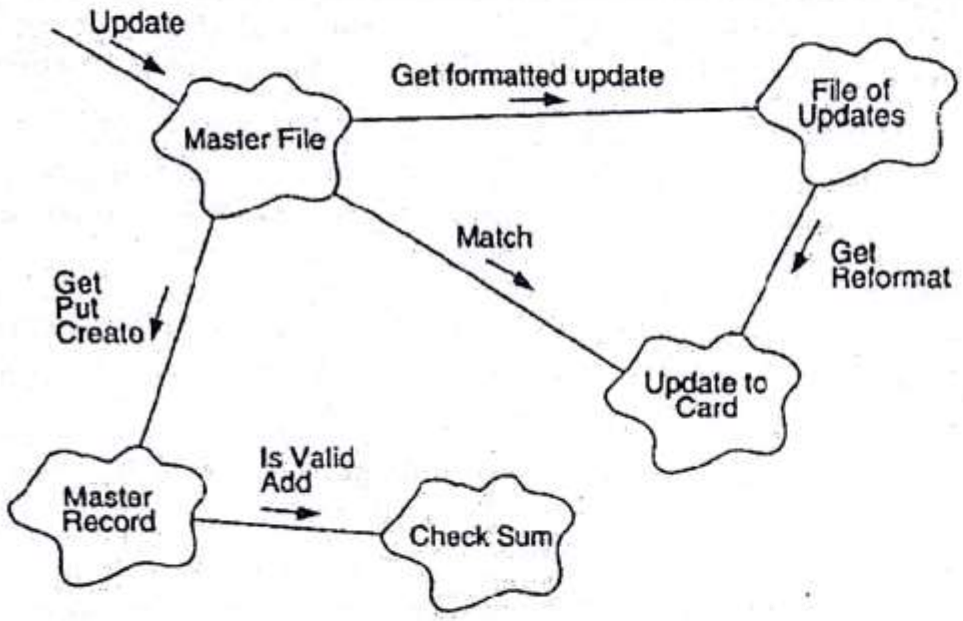


Figure 1.3: Object oriented decomposition

c. **Algorithmic versus object oriented decomposition:** Object oriented algorithm has a number of advantages over algorithmic decomposition. 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.

Although both solutions help dealing with complexity we have 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.

1.6 On Designing Complex Systems

Engineering as a Science and an Art: Every engineering discipline involves elements of both science and art. 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. The role of the engineer as artist is particularly challenging when the task is to design an entirely new system.

The meaning of Design: 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. The purpose of design is to construct a system that.

1. Satisfies a given (perhaps) informal functional specification
2. Conforms to limitations of the target medium

Downloaded by TONY 562 (tony2802b@gmail.com)

3. Meets implicit or explicit requirements on performance and resource usage
4. Satisfies implicit or explicit design criteria on the form of the artifact
5. Satisfies restrictions on the design process itself, such as its length or cost, or the available for doing the design.

According to Stroustrup, the purpose of design is to create a clean and relatively simple internal structure, sometimes also called as architecture. A design is the end product of the design process.

The Importance of Model Building: 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. We evaluate each model under both expected and unusual situations and then after them when they fail to behave as we expect or desire. More than one kind of model is used on order to express all the subtleties of a complex system.

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.

The models of Object Oriented Development: 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. These models also over the spectrum of the important design decisions that we must consider in developing a complex system and so encourage us to craft implementations that embody the five attributes of well formed complex systems.

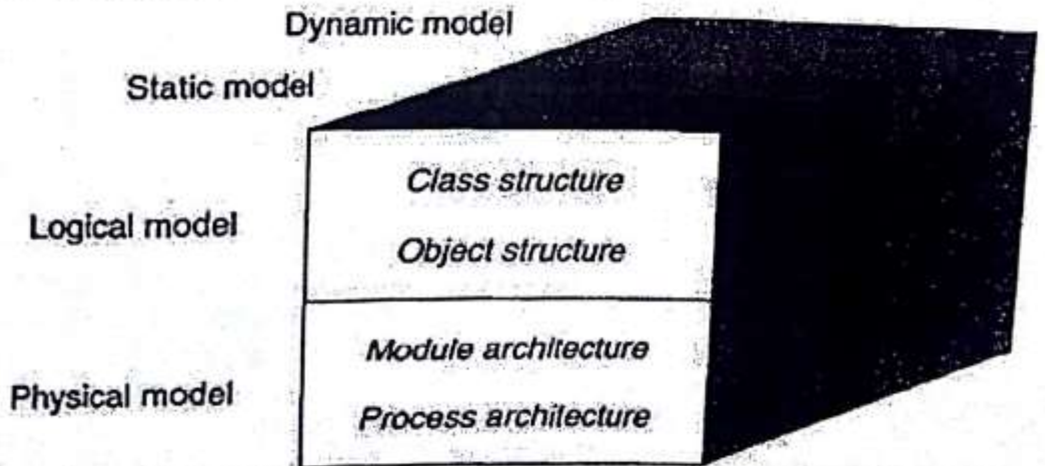


Figure 1.4: Models of object oriented development

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

Downloaded by TONY 562 (tony2802b@gmail.com)

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.

The Object Model

The elements of the object oriented technology collectively known as the object model. The object model encompasses the principles of abstraction, encapsulation, modularity, hierarchy, typing, concurrency and persistency. The object model brought together these elements in a synergistic way.

Downloaded by TONY 562 (tony2802b@gmail.com)

System Architecture: Satellite-Based Navigation

Here, 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.

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 sys-tem called Galileo is being developed by the European Union.

1. Inception

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 deter-mining 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.

Requirements for the Satellite Navigation System

Vision:

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

Downloaded by TONY 562 (tony2802b@gmail.com)

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

Downloaded by TONY 562 (tony2802b@gmail.com)

Defining the Boundaries of the Problem

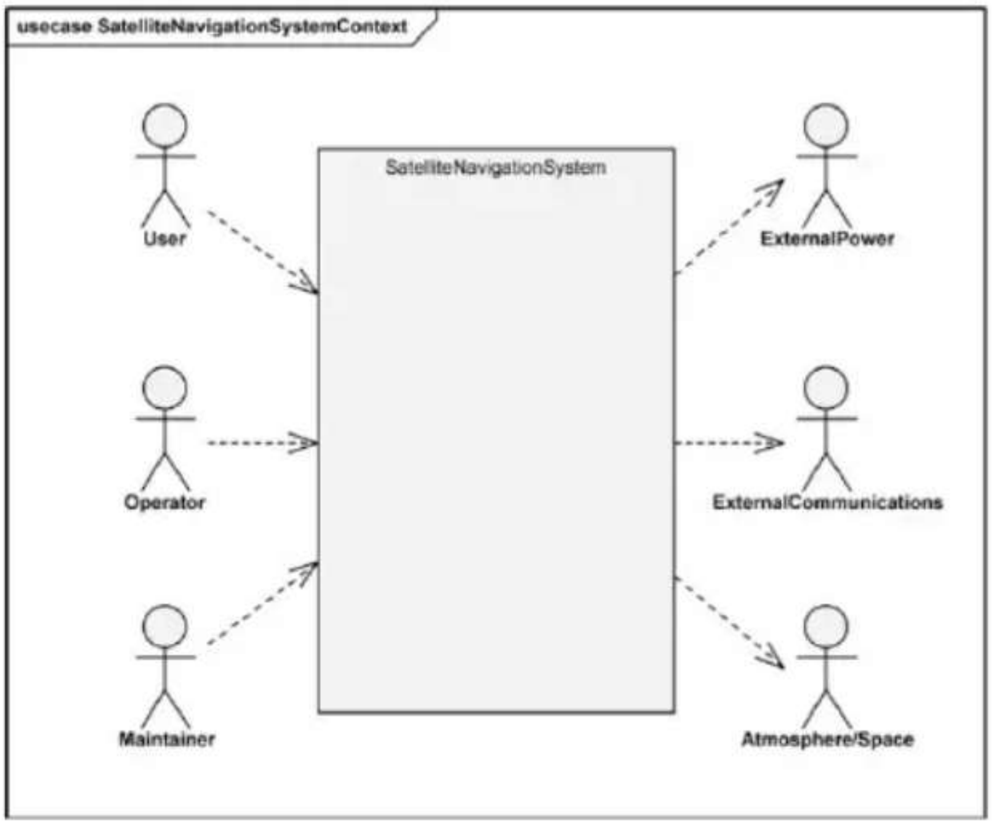


Figure 8–1 The Satellite Navigation System Context Diagram

Figure 8–1 The Satellite Navigation System Context Diagram

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.

Downloaded by TONY 562 (tony2802b@gmail.com)

Determining Mission Use Cases

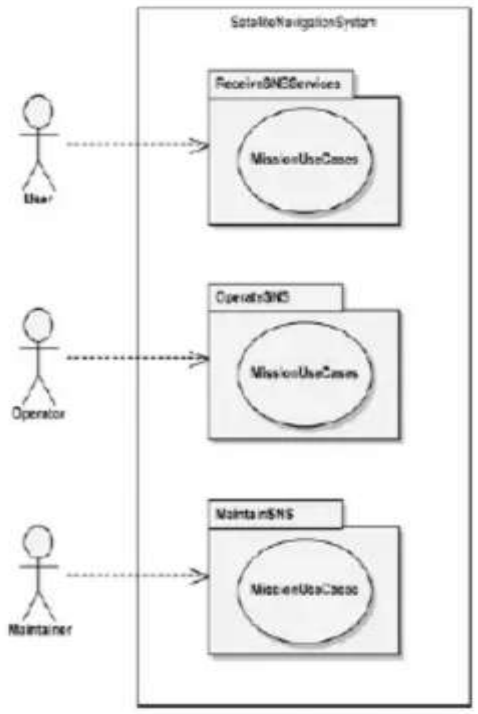


Figure 8–2 Packages for the SNS Mission Use Cases

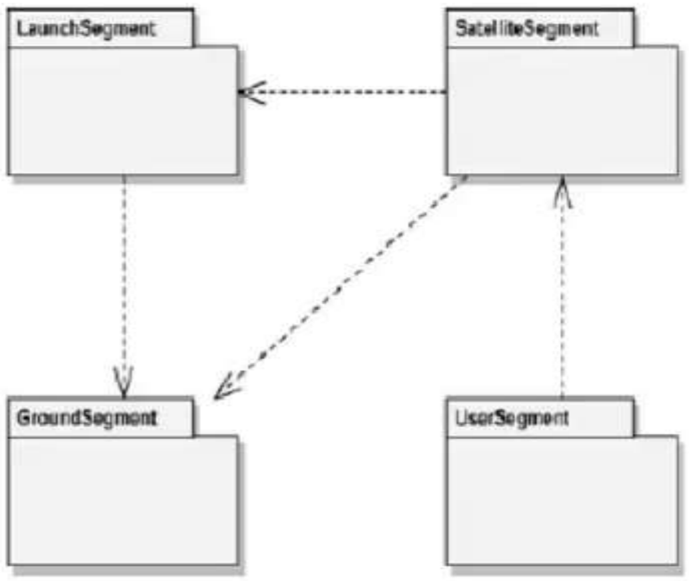


Figure 8–3 The SNS Logical Architecture

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 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.

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

Downloaded by TONY 562 (tony2802b@gmail.com)

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 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.

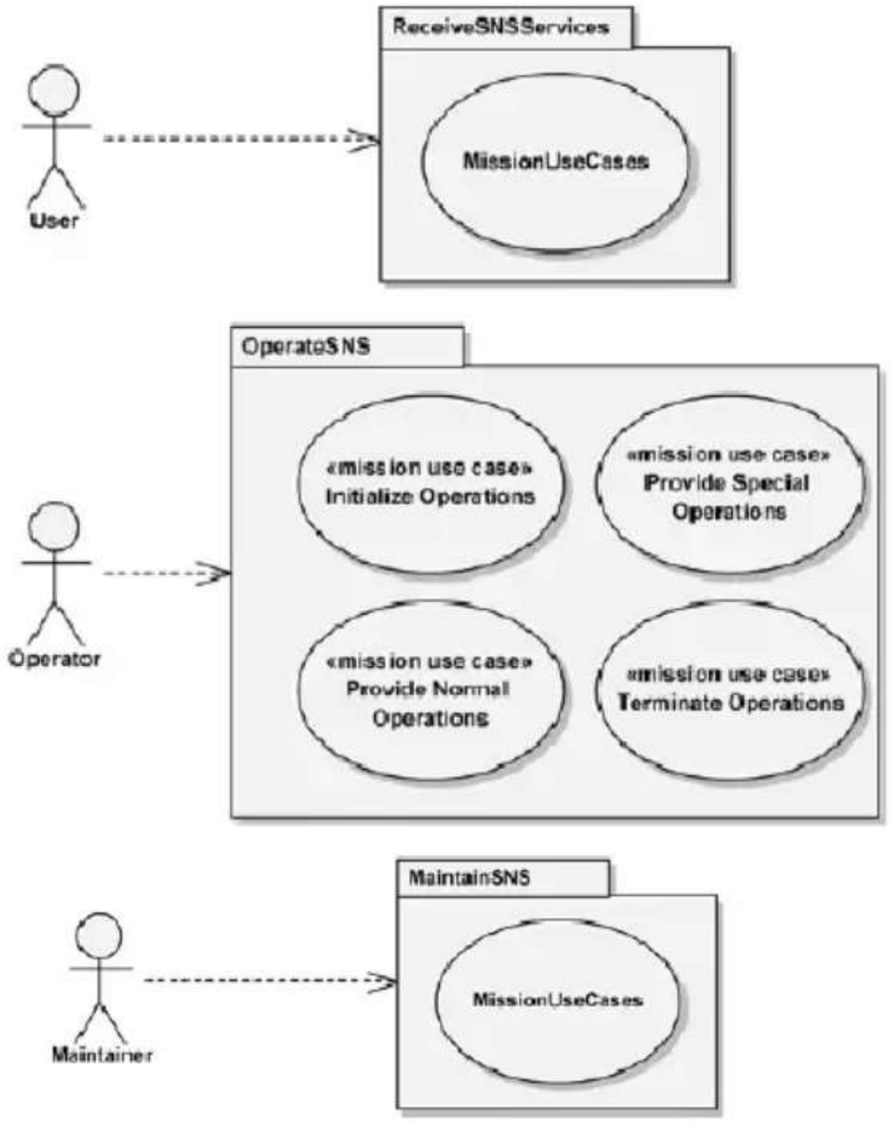

ReceiveSNSServices
User
MissionUseCases
OperateSNS
«mission use case» Initialize Operations
«mission use case» Provide Special Operations
Operator
«mission use case» Provide Normal Operations
«mission use case» Terminate Operations
MaintainSNS
Maintainer
MissionUseCases

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

Downloaded by TONY 562 (tony2802b@gmail.com)

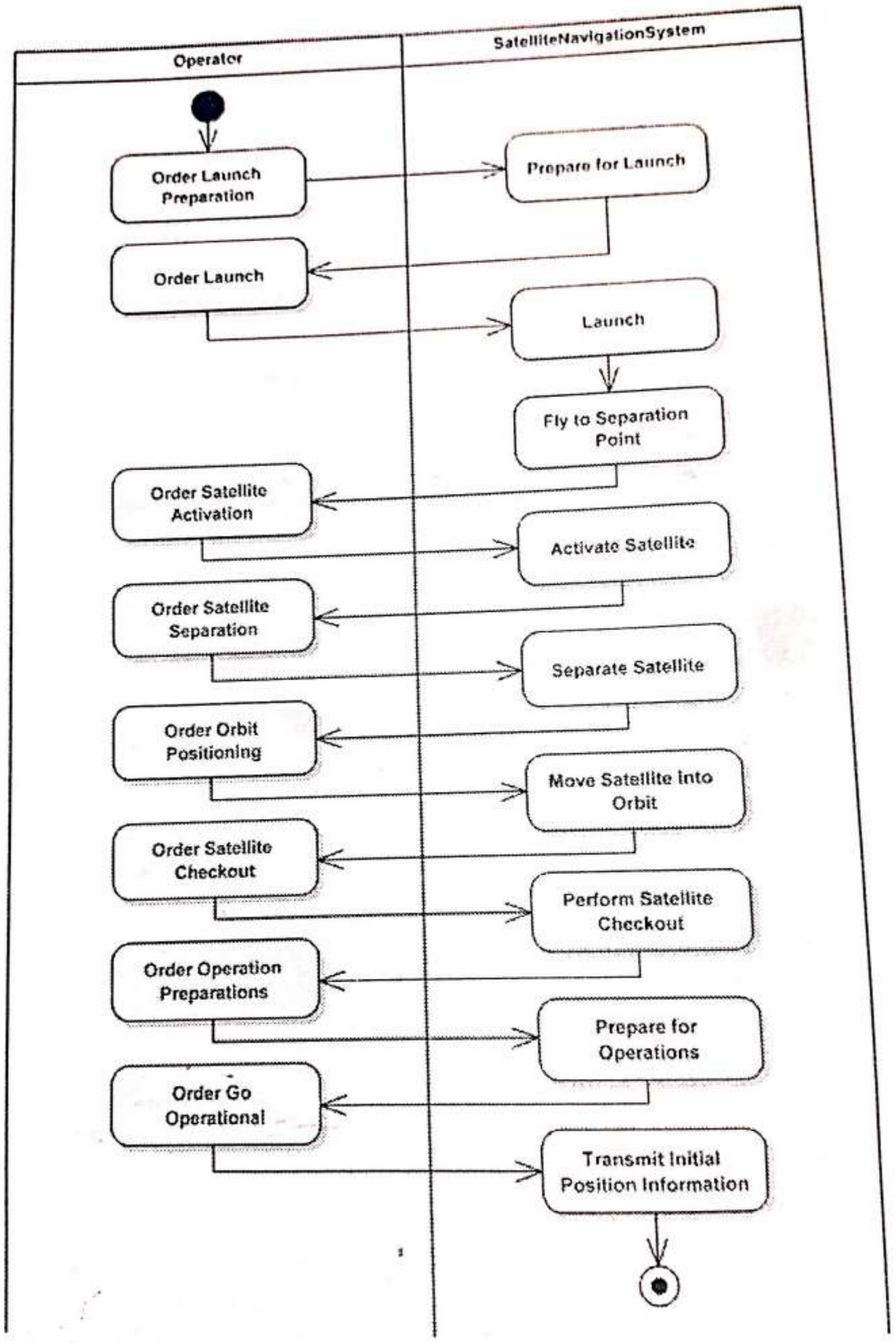
Determining System Use Cases

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.

Downloaded by TONY 562 (tony2802b@gmail.com)

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



Downloaded by TONY 562 (tony2802b@gmail.com)

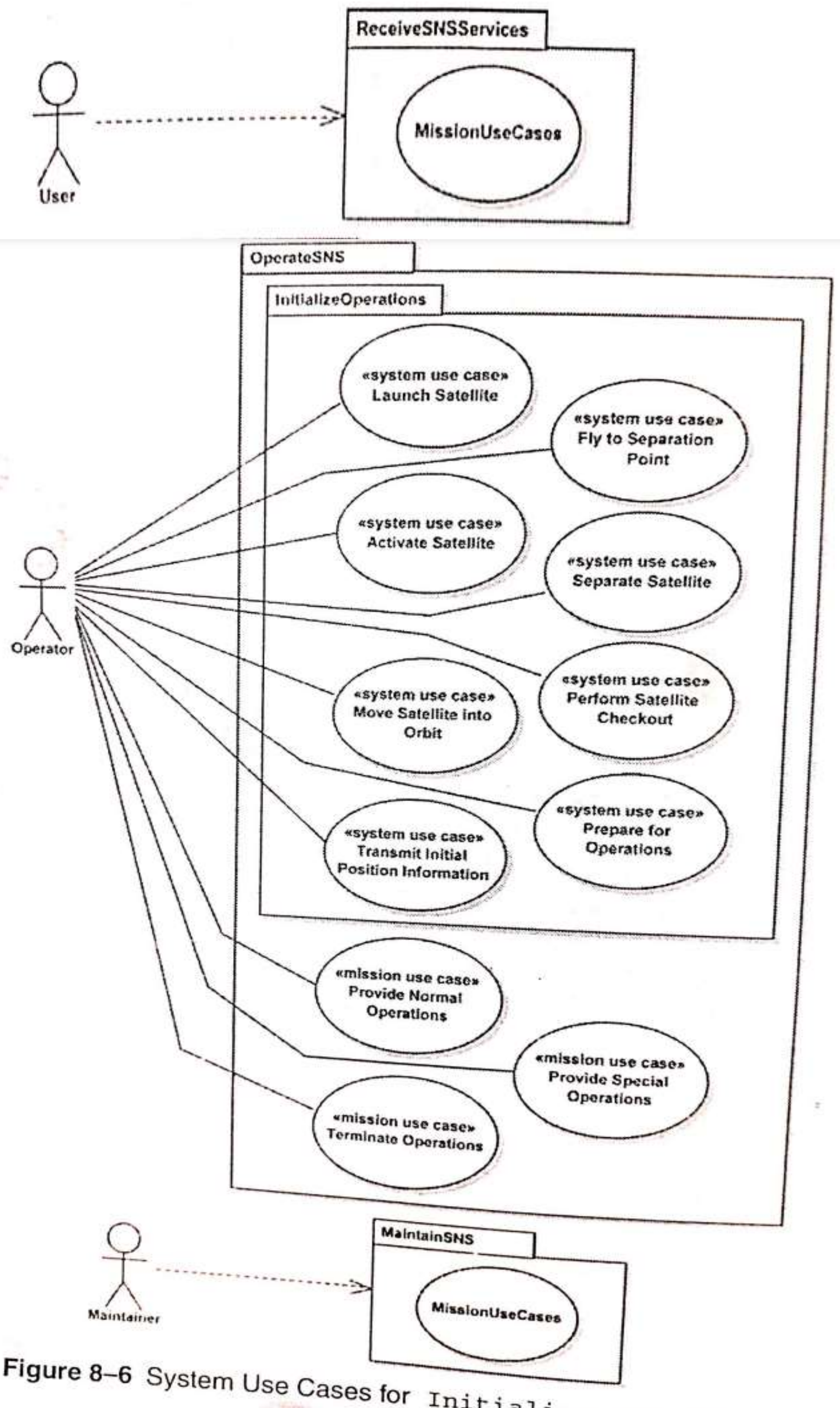
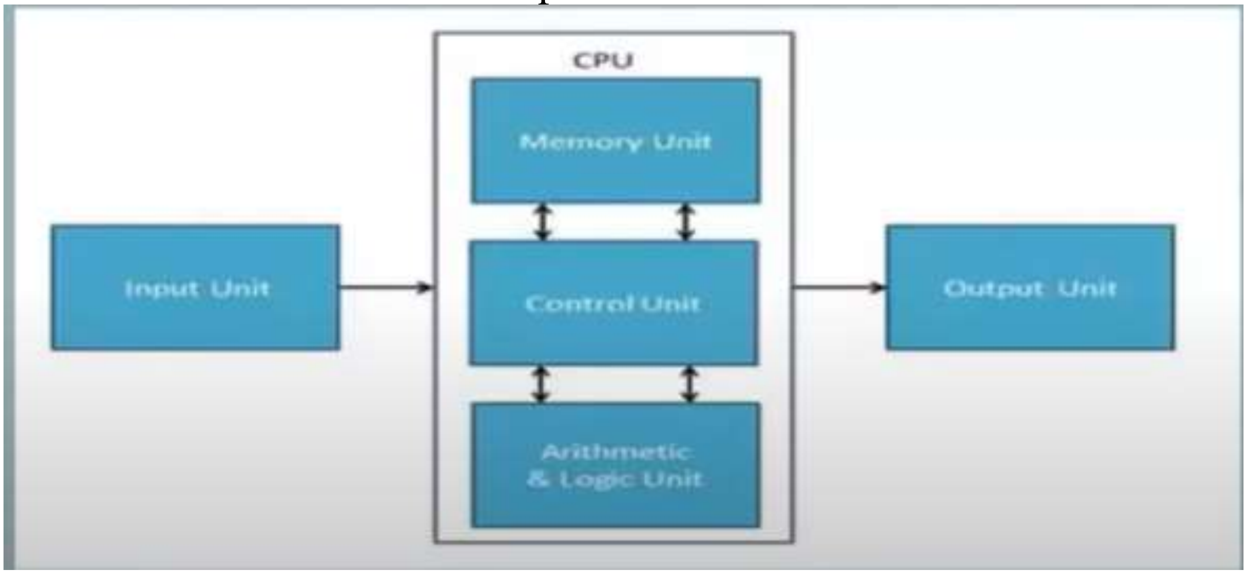

ReceiveSNSServices
MissionUseCases
User
OperateSNS
InitializeOperations
«system use case» Launch Satellite
«system use case» Fly to Separation Point
«system use case» Activate Satellite
«system use case» Separate Satellite
Operator
«system use case» Move Satellite into Orbit
«system use case» Perform Satellite Checkout
«system use case» Prepare for Operations
«system use case» Transmit Initial Position Information
«mission use case» Provide Normal Operations
«mission use case» Provide Special Operations
«mission use case» Terminate Operations
MaintainSNS
MissionUseCases
Maintainer

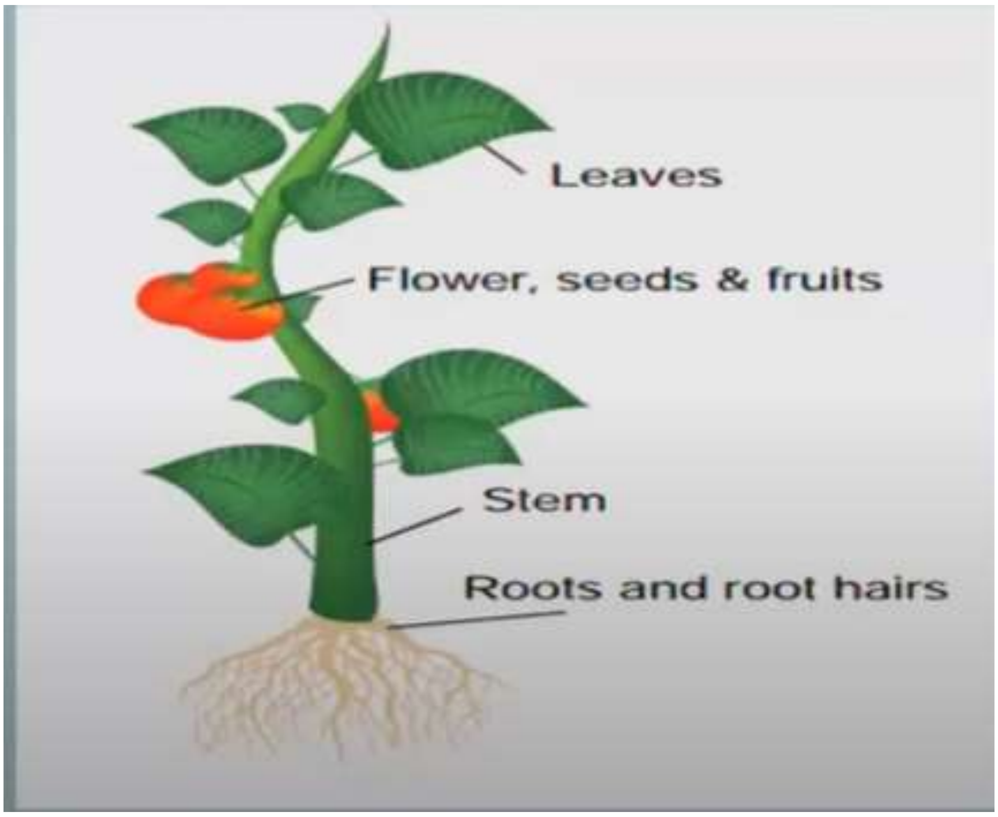
Figure 8–6 System Use Cases for `Initialize Operations`

Downloaded by TONY 562 (tony2802b@gmail.com)

The Structure of Personal Computer:

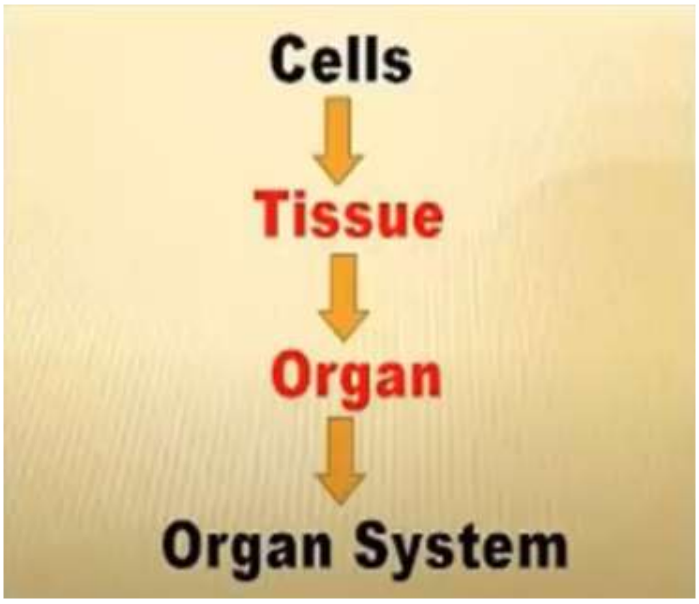


The Structure of Plants:

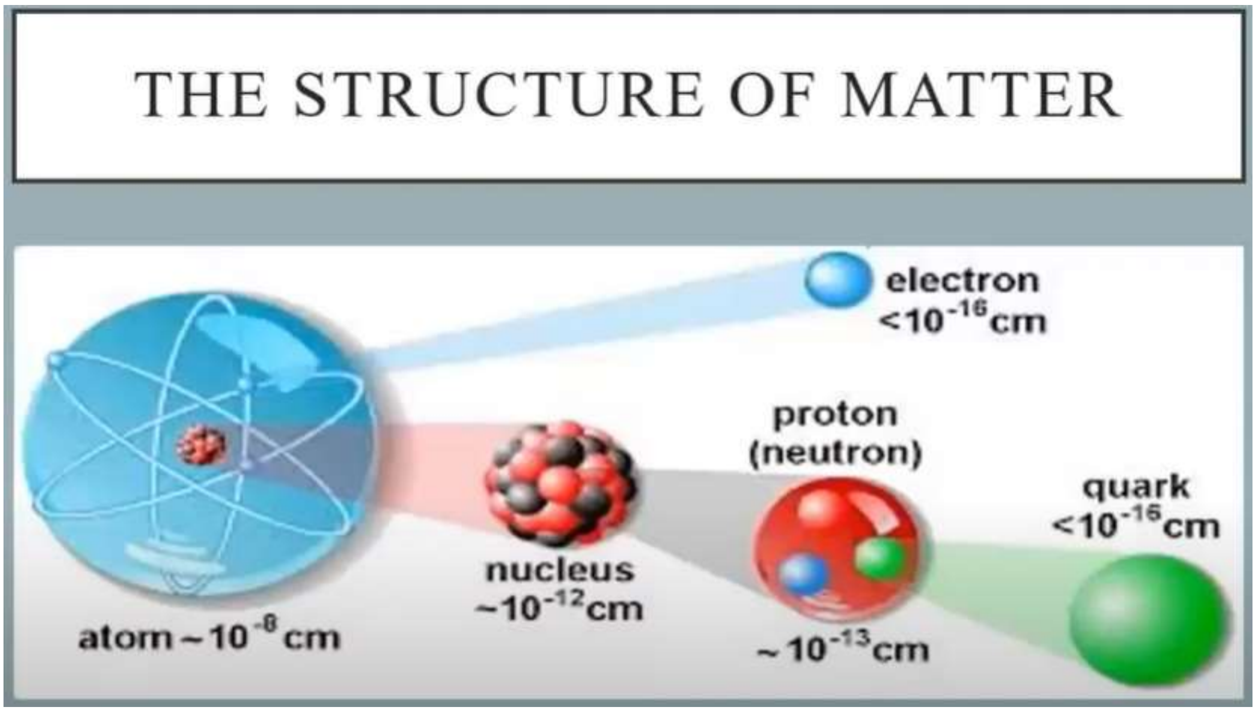


Downloaded by TONY 562 (tony2802b@gmail.com)

The Structure of Animals:

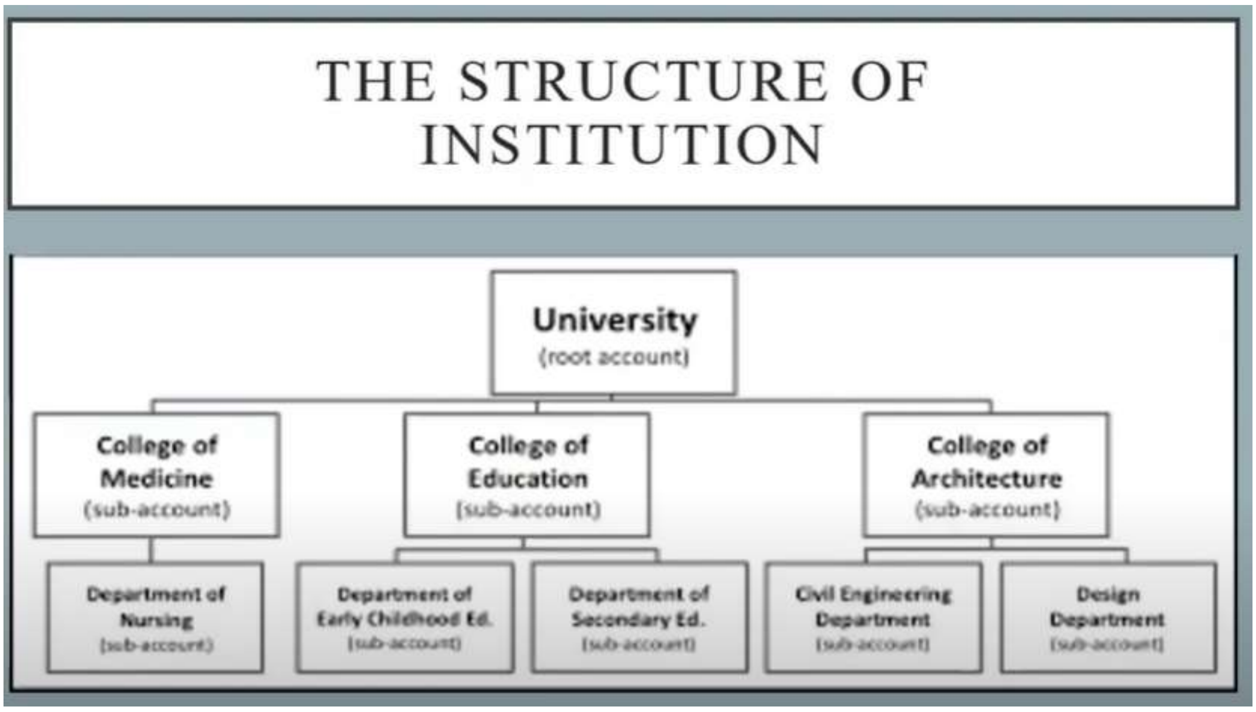


The Structure of Matter:



Downloaded by TONY 562 (tony2802b@gmail.com)

The Structure of Institution:


THE STRUCTURE OF INSTITUTION
University (root account)
College of Medicine (sub-account)
College of Education (sub-account)
College of Architecture (sub-account)
Department of Nursing (sub-account)
Department of Early Childhood Ed. (sub-account)
Department of Secondary Ed. (sub-account)
Civil Engineering Department (sub-account)
Design Department (sub-account)

*****THE END*****

Downloaded by TONY 562 (tony2802b@gmail.com)