LOVELY PROFESSIONAL UNIVERSITY

OBJECT ORIENTED ANALYSIS AND DESIGN

Edited By

Dr. Anil Sharma

Printed by
EXCEL BOOKS PRIVATE LIMITED
A-45, Naraina, Phase-I,
New Delhi-110028
for
Lovely Professional University
Phagwara

SYLLABUS

Object Oriented Analysis and Design

Objectives:

- To enable the student to understand the concept of Object Oriented Analysis and Design.
- To impart the skills required for modelling.
- To enable the student to implement generalization and inheritance class modeling.
- To enable the student to implement association and aggregation using modeling.
- To enable the student to implement state modeling
- To enable the student to implement interactive modeling.
- To enable the student to implement procedural and activity models.
- To impart the skills needed to perform system analysis and design.
- To enable the student to implement domain analysis.
- To enable the student to understand the technicalities of system design concepts.
- To enable the student to design class diagrams.

Sr. No.	Description
1.	***Introduction:*** object orientation, OO development, OO themes, Evidence for usefulness of OO development
2.	***Modelling concepts:*** Definition, Abstraction, Three models. ***Class Modelling:*** Object and class concepts, Link and association concepts, Generalization and inheritance, sample class model, Navigation of class models
3.	***Advance Class modelling:*** Advance object and class concepts, Association Ends, N-ary associations, aggregation, Abstract classes, Constraints, Derived data, packages.
4.	***State Modelling:*** Events, States, Transition and conditions, state diagrams, state diagram behaviour.
5.	***Interaction Modelling:*** Use case models, Sequence models, Activity models.
6.	***Advance Interaction Modelling:*** Use case Relationships, Procedural Sequence models, Special constructs for activity models
7.	***Analysis and design:*** process overview, development life cycle ***System conception:*** devising, Elaboration, Preparing a problem statement
8.	***Doman Analysis:*** Domain class model, Domain state model, domain interaction model.
9.	***System design:*** overview, reuse plan, concurrency, allocation, software control strategy, boundary conditions
10.	***Class design:*** Designing algorithms, Refactoring, design optimization, adjustment of inheritance, organizing class design.

CONTENT

Unit 1: Introduction to Object Orientation 1
Anil Sharma,Lovely Professional University

Unit 2: Modelling Concepts 13
Anil Sharma, Lovely Professional University

Unit 3: Class Modelling 25
Anil Sharma, Lovely Professional University

Unit 4: Advanced Class Modelling – I 38
Anil Sharma,Lovely Professional University

Unit 5: Advanced Class Modelling – II 51
Anil Sharma, Lovely Professional University

Unit 6: State Modelling 62
Anil Sharma, Lovely Professional University

Unit 7: Interaction Modelling 79
Anil Sharma, Lovely Professional University

Unit 8; Advanced Interaction Modelling 96
Anil Sharma, Lovely Professional University

Unit 9: Analysis and Design 109
Anil Sharma,Lovely Professional University

Unit 10: System Conception 118
Mandeep Kaur, Lovely Professional University

Unit 11: Domain Analysis 127
Mandeep Kaur, Lovely Professional University

Unit 12: System Design 140
Mandeep Kaur, Lovely Professional University

Unit 13: Class Design 155
Mandeep Kaur, Lovely Professional University

Unit 14: Steps for Class Design 166
Mandeep Kaur, Lovely Professional University

Unit 1: Introduction to Object Orientation

Notes

CONTENTS

Objectives

Introduction

1.1 Object Orientation (OO)

1.1.1 Uses of Object Orientation

1.1.2 Why Object Orientation?

1.2 Object-oriented Development

1.3 Object-oriented Themes

1.3.1 Abstraction

1.3.2 Encapsulation

1.3.3 Combining Data and Behaviour

1.3.4 Sharing

1.3.5 Emphasis on Object Structure, not on Operation Implementation

1.4 Evidence for Usefulness of Object-oriented Development

1.5 Summary

1.6 Keywords

1.7 Review Questions

1.8 Further Readings

Objectives

After studying this unit, you will be able to:

- Describe the concept of object orientation
- Discuss the uses and need of object orientation
- Explain the concept of object-oriented development
- Identify object-oriented themes
- Discuss the evidence for usefulness of object-oriented development

Introduction

Object-oriented Analysis and Design (OOAD) is a software engineering model which makes use of objects, classes, state, methods and behavior concepts to analyze and demonstrate system structure, functional needs and behavior. While object-oriented analysis emphasizes the things that a system actually does, the object oriented design is concerned with the manner in which the system does it. One thing that you must keep in mind is that the object oriented system will always be comprised of objects. The behavior for the system is a result for the connection that is made with the objects. Connections between objects will require them to send out messages to

Notes

one another. The object-oriented paradigm is currently the most popular way of analysing, designing, and developing application systems, especially large ones. Labelling something as 'object-oriented' implies that objects play a central role, and we elaborate this further as a perspective that views the elements of a given situation by decomposing them into objects and object relationships. In this unit, we will discuss the concept of object orientation and discuss object oriented development and object oriented themes.

1.1 Object Orientation (OO)

In the past, information systems used to be defined primarily by their functionality: data and functions were kept separate and linked together by means of input and output relations.

Object-oriented Approach means organizing software as a collection of discrete objects that incorporate both data structure and behavior. The object-oriented approach, however, focuses on objects that represent abstract or concrete things of the real world. These objects are first defined by their character and their properties which are represented by their internal structure and their attributes (data). The behaviour of these objects is described by methods (functionality).

Object orientation (OO), or to be more precise, object-oriented programming, is a problem-solving method in which the software solution reflects objects in the real world.

The term object-oriented (OO) signifies that we organize software as a collection of discrete objects that incorporate both data structure and behavior. This contrasts with previous programming approaches in which data structure and behavior are only loosely connected. There is some dispute regarding exactly what characteristics are needed by an object oriented approach, however they generally include aspects such as identity, classification, inheritance, polymorphism, etc.

Terms that are used universally in object orientation are:

- ***Objects:*** An object is a section of source code that contains data and provides services. The data forms the attributes of the object. The services are known as methods (also known as operations or functions). Typically, methods operate on private data (the attributes, or state of the object), which is only visible to the methods of the object. Thus the attributes of an object cannot be changed directly by the user, but only by the methods of the object. This guarantees the internal consistency of the object.

  Objects are a concept, abstraction, or item with clear boundaries and meaning for the problem domain.

  Objects have:

  - Unique identity
  - State
  - Behavior

  Objects can be concrete or conceptual, conceptual objects are pure abstractions that serve some specific purpose for a system. Objects are instances of classes. The set of activities that the object performs defines the object's behavior.

Example: A "StudentStatus" object can tell you its grade point average, year in school, or can add a list of courses taken. A "Student" object can tell you its name or its address.

- ***Classes:*** Classes describe objects. From a technical point of view, objects are runtime instances of a class. In theory, you can create any number of objects based on a single class.

Notes

A class or object class is a group of objects with similar properties (attributes), common behavior (operations), common relationships to other objects, and common semantics. A class is a definitive description of a group of objects with similar properties and behaviors. Classes are abstract, objects are concrete. Objects are aware of their class identity. Each instance (object) of a class has a unique identity and its own set of values for its attributes.

Objects within a class share a common semantic purpose above and beyond requirements of common attributes and behavior. The grouping of objects into classes abstracts a problem, that is, classes provide logical groupings of entities in a system. Individual objects within a class maintain their distinct identity.

- ***Object References:*** In a program, you identify and address objects using unique object references. Object references allow you to access the attributes and methods of an object.
- ***Identity*:** Identity means that data is quantized into discrete, distinguishable entities called objects. An object is a section of source code that contains data and provides services. Each object has its own inherent identity. In other words, two objects are distinct even if all their attribute values are identical.

Example: Name and size are the examples of attribute values.

In the real world an object simply exists, but within a programming language each object has a unique handle by which it can be referenced. Languages implement the handle in various ways such as an address, array index, or artificial number. Such object references are uniform and independent of the contents of the objects, permitting mixed collections of objects to be created, such as a file system directory that contains both files and sub-directories.

- ***Classification:*** Classification means that objects with the same data structure (attributes) and behavior (operations) are grouped into a class.

Example: Paragraph, Monitor, and ChessPiece are examples of classes.

A class is an abstraction that describes properties important to an application and ignores the rest. Any choice of classes is arbitrary and depends on the application.

Each class describes a possibly infinite set of individual objects. Each object is said to be an instance of its class. An object has its own value for each attribute but shares the attribute names and operations with other instances of the class. An object contains an implicit reference to its own class; it "knows what kind of thing it is."

- ***Polymorphism:*** Identical (identically-named) methods behave differently in different classes. Object-oriented programming contains constructions called interfaces. They enable you to address methods with the same name in different objects. Although the form of address is always the same, the implementation of the method is specific to a particular class.

Polymorphism is the kindred to the incorporation of data and behavior. The same operation may apply to many different classes, with some classes having distinct behavior (implementation). Such an object is polymorphic.

Example: A "draw" operation has different implementations for the classes circle, square, and polygon.

An operation is simply an abstraction of analogous behavior across different kinds of objects.

Notes

Each object "knows how to perform its operation." This burden is on the class and not the caller of the operation.

OOP selects the method to implement an operation based on the name and class of the operation. The user of an operation need not be aware of implementation details.

- ***Inheritance:*** Inheritance is a mechanism for sharing similarities among classes while preserving their differences. Classes are typically defined broadly (higher level of abstraction) and then refined into subclasses (lower level of abstraction).

  Each subclass "inherits" all of the properties of the superclass and adds its own unique properties. You can use an existing class to derive a new class. Derived classes inherit the data and methods of the superclass. However, they can overwrite existing methods, and also add new ones. Properties (both data and behavior) of the superclass need not be repeated in the subclass.

  The ability to abstract common properties of several classes into one common superclass and inherit the properties from the superclass can greatly reduce repetition within designs and programs (at the cost of coupling).

Example: Class 'employee' and class 'customer' are derived from the class 'person'

New classes can be added without changing code given that methods are provided for each applicable operation on the new class. Inheritance is used to model an "is a" relationship. A subclass is always an instance of the superclass.

Caution Be careful, the reverse is not true.

Task Explain the concept of "Identity" with example.

1.1.1 Uses of Object Orientation

Below are some of the advantages of object-oriented programming:

- Complex software systems become easier to understand, since object-oriented structuring provides a closer representation of reality than other programming techniques.
- In a well-designed object-oriented system, it should be possible to implement changes at class level, without having to make alterations at other points in the system. This reduces the overall amount of maintenance required.
- Through polymorphism and inheritance, object-oriented programming allows you to reuse individual components.
- In an object-oriented system, the amount of work involved in revising and maintaining the system is reduced, since many problems can be detected and corrected in the design phase.

Achieving these goals requires:

- ***Object-oriented programming languages:*** Object-oriented programming techniques do not necessarily depend on object-oriented programming languages. However, the efficiency

Notes

of object-oriented programming depends directly on how object-oriented language techniques are implemented in the system kernel.

- ***Object-oriented tools:*** Object-oriented tools allow you to create object-oriented programs in object-oriented languages. They allow you to model and store development objects and the relationships between them.
- ***Object-oriented modelling:*** The object-orientation modeling of a software system is the most important, most time-consuming, and most difficult requirement for attaining the above goals. Object-oriented design involves more than just object-oriented programming, and provides logical advantages that are independent of the actual implementation.

1.1.2 Why Object Orientation?

To create sets of objects that work together concurrently to produce s/w that better, model their problem domain that similarly system produced by traditional techniques.

It adapts to:

1. Changing requirements
2. Easier to maintain
3. More robust
4. Promote greater design
5. Code reuse
   (a) Higher level of abstraction
   (b) Seamless transition among different phases of software development
   (c) Encouragement of good programming techniques
   (d) Promotion of reusability

Self Assessment

Fill in the blanks:

1. ..... is a problem-solving method in which the software solution reflects objects in the real world.
2. An ..... is a section of source code that contains data and provides services. The data forms the attributes of the object.
3. Objects are runtime instances of a .....
4. ..... means that data is quantized into discrete, distinguishable entities called objects.
5. ..... means that objects with the same data structure (attributes) and behavior (operations) are grouped into a class.
6. The process in which same operation may apply to many different classes, with some classes having distinct behavior is known as .....
7. ..... is a mechanism for sharing similarities among classes while preserving their differences.
8. Object-oriented ..... allow you to create object-oriented programs in object-oriented languages.

Notes

1.2 Object-oriented Development

Object Oriented Development is a new way of thinking about software based on abstractions that exist in the real world as well as in the program. Object Oriented Development is a method of design encompassing the process of object-oriented decomposition and a notation for depicting logical and physical as well as static and dynamic models of the system under design.

In this context development refers to the software life cycle; analysis, design, and implementation. The essence of OO development is the identification and organization of application concepts, rather than their final representation in a programming language. It is observed that the hard part of software development is the manipulation of its essence, owing to the inherent complexity of the problem, rather than the accidents of its mapping into a particular language.

The traditional view of a computer program is that of a process that has been encoded in a form that can be executed on a computer. This view originated from the fact that the first computers were developed mainly to automate a well-defined process (i.e., an algorithm) for numerical computation, and dates back to the first stored-program computers. Accordingly, the software creation process was seen as a translation from a description in some 'natural' language to a sequence of operations that could be executed on a computer. As many would argue, this paradigm is still the best way to introduce the notion of programming to a beginner, but as systems became more complex, its effectiveness in developing solutions became suspect. This change of perspective on part of the software developers happened over a period of time and was fuelled by several factors including the high cost of development and the constant efforts to find uses for software in new domains. One could safely argue that the software applications developed in later years had two differentiating characteristics:

- Behaviour that was hard to characterize as a process
- Requirements of reliability, performance, and cost that the original developers did not face

The 'process-centred' approach used to software development is called top-down functional decomposition. The first step in such a design was to recognise what the process had to deliver (in terms of input and output of the program), which was followed by decomposition of the process into functional modules. Structures to store data were defined and the computation was carried out by invoking the modules, which performed some computation on the stored data elements. The life of a process-centred design was short because changes to the process specification (something relatively uncommon with numerical algorithms when compared with business applications) required a change in the entire program. This in turn resulted in an inability to reuse existing code without considerable overhead. As a result, software designers began to scrutinise their own approaches and also study design processes and principles that were being employed by engineers in other disciplines. Cross-pollination of ideas from other engineering disciplines started soon after, and the disciplines of 'software design' and 'software engineering' came into existence.

In this connection, it is interesting to note the process used for designing simple electromechanical systems. For several decades now, it has been fairly easy for people with limited knowledge of engineering principles to design and put together simple systems in their backyards and garages. So much so, it has become a hobby that even a ten-year old could pursue. The reasons for this success are easy to see:

- easily understandable designs
- similar (standard) solutions for a host of problems
- an easily accessible and well-defined 'library' of 'building-blocks'
- interchangeability of components across systems, etc.

Notes

Some of the pioneers in the field of software design began to ask whether they could not also design software using such 'off-the-shelf' components. The object-oriented paradigm one could argue has really evolved in response to this outlook. There are, of course, several differences with the hardware design process (inevitable, because the nature of software is fundamentally different from hardware), but parallels can be drawn between many of the defining characteristics of hardware design and what today's advocates of good software design recommend. This methodology provides us with a step-by-step process for software design, a language to specify the output from each step of the process so that we can transition smoothly from one stage to the next, the ability to reuse earlier designs, standard solutions that adhere to well-reasoned design principles and, even the ability to incrementally fix a poor design without breaking the system.

The overall philosophy here is to define a software system as a collection of objects of various types that interact with each other through well-defined interfaces. Unlike a hardware component, a software object can be designed to handle multiple functions and can therefore participate in several processes.

Notes A software component is also capable of storing data, which adds another dimension of complexity to the process.

The manner in which all of this has departed from the traditional process-oriented view is that instead of implementing an entire process end-to-end and defining the needed data structures along the way, we first analyse the entire set of processes and from this identify the necessary software components. Each component represents a data abstraction and is designed to store information along with procedures to manipulate the same. The execution of the original processes is then broken down into several steps, each of which can be logically assigned to one of the software components.

Did u know? The components can also communicate with each other as needed to complete the process.

Self Assessment

Fill in the blanks:

9. ..... is a new way of thinking about software based on abstractions that exist in the real world as well as in the program.
10. ..... system is defined as a collection of objects of various types that interact with each other through well-defined interfaces.

1.3 Object-oriented Themes

There are several themes in an object oriented technology. These themes are not unique to object oriented systems. We can see some important themes:

1.3.1 Abstraction

Abstraction consists of focusing on the essential, inherent aspects of an entity and ignoring its accidental aspects. Use of abstraction preserves the freedom to make decisions as long as possible by avoiding premature commitments to details. Most modern languages provide data abstraction,

Notes

but inherit-mice and polymorphism add power. The ability to abstract is probably the most important skill required for object oriented development.

1.3.2 Encapsulation

Encapsulation is the process of encapsulating the elements of an abstraction that constitutes its structure and behavior; encapsulation serves to separate the contractual interface of an abstraction and its implementation.

It can also be called as information hiding. It consists of separating the external aspects of an object, which are accessible to other objects, from the internal implementation details of the object, which are hidden from other objects. It is not unique to object oriented languages.

Encapsulation (also information hiding) consists of separating the external aspects of an object, which are accessible to other objects, from the internal details of the object, which are hidden from other objects.

Objects have an outside (how they are seen or interact) and an inside (what they are.)

Encapsulation means that every object is self-contained.

Encapsulation is not a unique concept to OO languages, but the ability to combine data and behavior in an object provides cleaner encapsulations than with conventional languages.

Objects restrict the visibility of their resources (attributes and methods) to other users. Every object has an interface, which determines how other objects can interact with it. The implementation of the object is encapsulated, that is, invisible outside the object itself.

Separate the external aspects of an object (its interface) from the internal implementation details (implementation).

Proper encapsulation keeps objects from becoming interdependent (or coupled) - internal details can change with no affect on the user. Encapsulation places a premium on interfaces.

Caution Encapsulated details can change with no impact to a client, while an interface change directly affects a client.

1.3.3 Combining Data and Behaviour

The caller of an operation need not consider how many implementations of a given operation exist. Operator polymorphism shifts the burden of deciding what implementation to use from the calling code to the class hierarchy.

As an example, let us talk about an object oriented program calling a draw procedure for drawing different figures say a polygon, circle, or text. The decision of which procedure to use is made by each object, based on its class.

1.3.4 Sharing

Inheritance of both data structure and behavior allows common structure to be shared among several similar subclasses without redundancy. The sharing of code using inheritance is one of the main advantages of object oriented languages.

One of the reasons for the popularity of object-oriented techniques is that they encourage sharing at different levels. Inheritance of both data structure and behavior allows common structure (base class) to be used in designing many subclasses based on basic characteristics of base class,

Notes

and develop new classes with less effort. Inheritance is one of the main advantages of any object oriented language, because it gives scope to share basic code.

In a broader way we can say that object oriented development not only allows information sharing and reuse within an application, but also, it gives a base for project enhancement in future. As and when there is a need for adding new characteristics in the system, they can be added as an extension of existing basic features. This can be done by using inheritance and that too, without major modification in the existing code. But be aware that just by using object orientation, you do not get a license to ensure reusability and enhancement. For ensuring reusability and enhancement you have to have a more general design of the system.

Did u know? This type of design can be developed only if the system is properly studied and features of proposed system are explored.

Task How does sharing of code using inheritance prove to be an advantage of object oriented languages?

1.3.5 Emphasis on Object Structure, not on Operation Implementation

In object orientation the major emphasis is on specifying the characteristics of the objects in a system, rather than implementing these characteristics. The uses of an object depend highly on the facts of the application and regular changes during development. As requirements extend, the features supplied by an object are much more stable than the ways in which they are used, hence software systems built on object structure are more secure.

While developing a system using the object oriented approach, main emphasis is on the essential properties of the objects involved in the system than on the procedure structure to be used for implementation. During this process what an object is, and its role in system is deeply thought about.

Self Assessment

Fill in the blanks:

11. ..... consists of focusing on the essential, inherent aspects of an entity and ignoring its accidental aspects.
12. The process of encapsulation is also known as .....
13. Inheritance of both data structure and behavior allows common structure to be shared among several similar subclasses without .....

1.4 Evidence for Usefulness of OO Development

Object oriented development began at the General Electric Research and Development Center. We used object oriented technologies for developing compilers, graphics, user interfaces, databases, an object oriented language, CAD systems, simulations, meta models, control systems, and other applications. We used object oriented models to document programs that are ill-structured and difficult to understand. Our implementation targets ranged from object oriented languages to non object-oriented languages to databases. We successfully taught this approach to others and used it to communicate with application experts.

Notes

Since the mid 1990s, we have expanded our practice of object oriented technology beyond General Electric to companies throughout the world. Earlier object orientation and object oriented modeling were relatively new approaches without much large-scale experience.

Notes Object oriented technology can no longer be considered a fad or a speculative approach. It is now part of the computer science and software engineering mainstream.

The annual OOPSLA (Object-oriented Programming Systems, Languages, and Applications), ECOOP (European Conference on Object-oriented Programming), and TOOLS (Technology of Object-oriented Languages and Systems) conferences are important forums for disseminating new object oriented ideas and application results. The conference proceedings describe many applications that have benefited from an object oriented approach.

Advantages of OO Development are:

- Reusability
- Effective maintenance

Disadvantage of OO Development:

- Not applicable in performance critical rather than data.

Self Assessment

State whether the following statements are true or false:

14. Object oriented development began at the General Electric Research and Development Center.
15. Object oriented models were used to document programs that are well-structured and easy to understand.

1.5 Summary

- Object-oriented analysis and design is a software engineering model which makes use of objects, classes, state, methods and behavior concepts to analyze and demonstrate system structure, functional needs and behavior.
- Object Oriented Approach means organizing software as a collection of discrete objects that incorporate both data structure and behavior.
- Object orientation (OO), or to be more precise, object-oriented programming, is a problem-solving method in which the software solution reflects objects in the real world.
- An object is a section of source code that contains data and provides services.
- A class is a definitive description of a group of objects with similar properties and behaviors.
- Object Oriented Development is a new way of thinking about software based on abstractions that exist in the real world as well as in the program.
- Abstraction consists of focusing on the essential, inherent aspects of an entity and ignoring its accidental aspects.
- Encapsulation is the process of encapsulating the elements of an abstraction that constitutes its structure and behavior.

Notes

- Inheritance of both data structure and behavior allows a common structure and behavior allows common structure to be shared among several similar subclasses without redundancy.
- In object orientation the major emphasis is on specifying the characteristics of the objects in a system, rather than implementing these characteristics.

1.6 Keywords

Abstraction: Abstraction consists of focusing on the essential, inherent aspects of an entity and ignoring its accidental aspects.

Class: A class or object class is a group of objects with similar properties (attributes), common behavior (operations), common relationships to other objects, and common semantics.

Classification: Classification means that objects with the same data structure (attributes) and behavior (operations) are grouped into a class.

Encapsulation: Encapsulation is the process of encapsulating the elements of an abstraction that constitutes its structure and behavior.

Identity: Identity means that data is quantized into discrete, distinguishable entities called objects.

Inheritance: Inheritance is a mechanism for sharing similarities among classes while preserving their differences.

Object: An object is a section of source code that contains data and provides services.

OO Development: Object-oriented Development is a new way of thinking about software based on abstractions that exist in the real world as well as in the program.

1.7 Review Questions

1. What is object orientation? Explain with example.
2. Describe the terms related to object orientation. Give example of each.
3. Make distinction between polymorphism and inheritance.
4. What are the advantages of object orientation? Discuss.
5. Discuss the importance of object orientation.
6. Elucidate the concept of object-oriented development.
7. What are the different themes in an object-oriented technology? Discuss.
8. "In object orientation the major emphasis is on specifying the characteristics of the objects in a system, rather than implementing these characteristics." Comment.
9. Write short note on the evidence for usefulness of OO Development.
10. Explain the concept of encapsulation with example.

Answers: Self Assessment

1.	Object orientation (OO)	2.	Object
3.	class	4.	Identity
5.	Classification	6.	polymorphism.

Notes

7. Inheritance
8. Tools
9. Object Oriented Development
10. Software
11. Abstraction
12. information hiding
13. redundancy
14. True
15. False

1.8 Further Readings

Books

Booch, Grady, (1994), *Object-oriented Analysis & Design*, Addison Wesley

Laganiere, Robert, (2004), *Object-oriented Software Engineering*, TMH

Rumbaugh, J., (2007), *Object-oriented Modelling and Design with UML*, Pearson Education

Satzinger, (2007), *Object-oriented Analysis & Design with the Unified Process*, Thomson

Online links

http://dl.acm.org/citation.cfm?id=13433

http://help.sap.com/saphelp_nw70/helpdata/en/c3/225b5654f411d194a60000e8353423/content.htm

http://www.ics.uci.edu/~taylor/classes/121/BoochOOD003.pdf

http://www.trosolwg.co.uk/What_are_objects.htm

Notes

Unit 2: Modelling Concepts

CONTENTS

Objectives

Introduction

2.1 Basics of Object-oriented Analysis and Design (OOAD)

2.1.1 Object-oriented Analysis (OOA)

2.1.2 Object-oriented Design (OOD)

2.2 Modelling

2.2.1 Definition of Modelling

2.2.2 Why do we Model?

2.2.3 Object-oriented Modelling (OOM)

2.2.4 Benefits of Object-oriented Modelling

2.3 Abstraction

2.4 Three Models

2.5 Summary

2.6 Keywords

2.7 Review Questions

2.8 Further Readings

Objectives

After studying this unit, you will be able to:

- Describe the basics of Object-oriented Analysis and Design
- Discuss the concept of modelling
- Explain abstraction
- Discuss three models used in object-oriented modelling

Introduction

Object oriented design methods emerged in the 1980s, and object oriented analysis methods emerged during the 1990s. In the early stage, object orientation was largely associated with the development of Graphical User Interfaces (GUIs), and a few other applications became widely known. In the 1980s, Grady Booch published a paper on how to design for Ada and gave it the title, Object Oriented Design. In 1991, Booch was able to extend his ideas to a genuinely object oriented design method with the same title, revised in 1993. The Object Modelling Technique (OMT) covers aspects of object oriented analysis and design. OOT provides a very productive and practical way of software development. As Object-oriented Technology (OOT) is not language dependent, there is no need for considering a final implementation language, during Object-oriented Modelling (OOM). OOT combines structural, control and functional aspects of the system.

Notes

2.1 Basics of Object-oriented Analysis and Design (OOAD)

Object-oriented Analysis and Design is a software engineering approach that models a system as a group of interacting objects. Each object represents some entity of interest in the system being modeled, and is characterized by its class, its state (data elements), and its behavior. Various models can be created to show the static structure, dynamic behavior, and run-time deployment of these collaborating objects. There are a number of different notations for representing these models, such as the Unified Modeling Language (UML).

Object-oriented Analysis (OOA) applies object-modeling techniques to analyze the functional requirements for a system. Object-oriented Design (OOD) elaborates the analysis models to produce implementation specifications.

OOA focuses on what the System Does, OOD on How the System Does it. An object-oriented system is composed of objects. The behavior of the system results from the collaboration of those objects. Collaboration between objects involves them sending messages to each other. Sending a message differs from calling a function in that when a target object receives a message, it itself decides what function to carry out to service that message. The same message may be implemented by many different functions, the one selected depending on the state of the target object.

The implementation of "message sending" varies depending on the architecture of the system being modeled, and the location of the objects being communicated with.

2.1.1 Object-oriented Analysis (OOA)

Object-oriented Analysis looks at the problem domain, with the aim of producing a conceptual model of the information that exists in the area being analyzed. Analysis models do not consider any implementation constraints that might exist, such as concurrency, distribution, persistence, or how the system is to be built. Implementation constraints are dealt with during Object-oriented Design. Analysis is done before the Design.

The sources for the analysis can be a written requirements statement, a formal vision document, and interviews with stakeholders or other interested parties.

Did u know? A system may be divided into multiple domains, representing different business, technological, or other areas of interest, each of which are analyzed separately.

The result of object-oriented analysis is a description of what the system is functionally required to do, in the form of a conceptual model. That will typically be presented as a set of use cases, one or more UML class diagrams, and a number of interaction diagrams. It may also include some kind of user interface mock-up.

2.1.2 Object-oriented Design (OOD)

Object-oriented Design transforms the conceptual model produced in object-oriented analysis to take account of the constraints imposed by the chosen architecture and any non-functional – technological or environmental – constraints.

Examples of non-functional constraints include transaction throughput, response time, run-time platform, development environment, or programming language.

The concepts in the analysis model are mapped onto implementation classes and interfaces. The result is a model of the solution domain, a detailed description of how the system is to be built.

Notes

Self Assessment

Fill in the blanks:

1. ..... looks at the problem domain, with the aim of producing a conceptual model of the information that exists in the area being analyzed.
2. ..... elaborates the analysis models to produce implementation specifications.
3. The concepts in the ..... model are mapped onto implementation classes and interfaces.

2.2 Modelling

In this section, we will discuss the concept of modelling.

2.2.1 Definition of Modelling

A model is an abstraction of something for the purpose of understanding it before building it. Because, real systems that we want to study are generally very complex. In order to understand the real system, we have to simplify the system. So a model is an abstraction that hides the non-essential characteristics of a system and highlights those characteristics, which are pertinent to understand it. Efraim Turban describes a model as a simplified representation of reality. A model provides a means for conceptualization and communication of ideas in a precise and unambiguous form. The characteristics of simplification and representation are difficult to achieve in the real world, since they frequently contradict each other. Thus modeling enables us to cope with the complexity of a system.

Most modeling techniques used for analysis and design involve graphic languages. These graphic languages are made up of sets of symbols. As you know one small line is worth thousand words. So, the symbols are used according to certain rules of methodology for communicating the complex relationships of information more clearly than descriptive text.

Modeling is used frequently, during many of the phases of the software life cycle such as analysis, design and implementation. Modeling like any other object-oriented development, is an iterative process. As the model progresses from analysis to implementation, more detail is added to it.

2.2.2 Why do we Model?

Before constructing anything, a designer first build a model. The main reasons for constructing models include:

- To test a physical entity before actually building it.
- To set the stage for communication between customers and developers.
- For visualization i.e. for finding alternative representations.
- For reduction of complexity in order to understand it.

2.2.3 Object-oriented Modelling (OOM)

Object oriented modeling is entirely a new way of thinking about problems. This methodology is all about visualizing the things by using models organized around real world concepts. Object oriented models help in understanding problems, communicating with experts from a

Notes

distance, modeling enterprises, and designing programs and database. We all can agree that developing a model for a software system, prior to its development or transformation, is as essential as having a blueprint for large building essential for its construction. Object oriented models are represented by diagrams. A good model always helps communication among project teams, and to assure architectural soundness.

It is important to note that with the increasing complexity of systems, importance of modeling techniques increases. Because of its characteristics Object Oriented Modeling is a suitable modeling technique for handling a complex system. OOM basically is building a model of an application, which includes implementation details of the system, during design of the system.

As you know, any system development refers to the initial portion of the software life cycle: analysis, design, and implementation. During object oriented modeling identification and organization of application with respect to its domain is done, rather than their final representation in any specific programming language. We can say that OOM is not language specific.

Notes Once modeling is done for an application, it can be implemented in any suitable programming language available.

OOM approach is an encouraging approach in which software developers have to think in terms of the application domain through most of the software engineering life cycle. In this process, the developer is forced to identify the inherent concepts of the application. First, developer organizes and understands the system properly and then finally the details of data structure and functions are addressed effectively.

Object-oriented modelling is a methodology of analyzing requirements of a system with the aim of identifying sub-systems with the following desirable properties:

(a) Each subsystem should have clearly specified responsibility of performing a part of overall task.

(b) Other parts of the sub-system should not have to know how a subsystem performs the task assigned to it, rather they should only know what task a subsystem does

(c) Each sub-system should be self-contained and independent

(d) Each sub-system should know what other subsystems do and how to send requests to them for assistance so that it can co-operate with them to get its own job done

(e) Sub-system should hide from outside world the data it uses

(f) The sub-system should be designed to be reusable

Object-oriented modelling is used in practice as it

- Facilitates changing of system to improve functionality during the system life time
- Facilitates reuse of code of each of the subsystems used to design the large system
- Facilitates integrating subsystems into a large system
- Facilitates design of distributed systems

An object-oriented modelling is particularly useful in the following situations:

- It is required to change an existing system by adding new functionality.
- While designing large system and it is found that it can be designed as a collection of existing reusable objects.

Notes

In OOM the modeling passes through the following processes:

- System Analysis,
- System Design,
- Object Design, and
- Final Implementation.

System Analysis: In this stage a statement of the problem is formulated and a model is build by the analyst in encouraging real-world situation. This phase show the important properties associated with the situation. Actually, the analysis model is a concise, precise abstraction and agreement on how the desired system must be developed. You can say that, here the objective is to provide a model that can be understood and criticized by any application experts in the area whether the expert is a programmer or not.

System Design: At this stage, the complete system architecture is designed. This is the stage where the whole system is divided into subsystems, based on both the system analysis model and the proposed architecture of the system.

Object Design: At this stage, a design model is developed based on the analysis model which is already developed in the earlier phase of development. The object design decides the data structures and algorithms needed to implement each of the classes in the system with the help of implementation details given in the analysis model.

Final Implementation: At this stage, the final implementation of classes and relationships developed during object design takes place a particular programming language, database, or hardware implementation (if needed).

Caution Actual implementation should be done using software engineering practice. This helps to develop a flexible and extensible system.

Task Make distinction between system design and object design.

2.2.4 Benefits of Object-oriented Modelling

There are several advantages and benefits of object oriented modeling. Reuse and emphasis on quality are the major highlights of OOM. OOM provides resistance to change, encapsulation and abstraction, etc. Due to its very nature, all these features add to the systems development:

- Faster development
- Increased quality
- Easier maintenance
- Reuse of software and designs, frameworks
- Reduced development risks for complex systems integration.

The conceptual structure of object orientation helps in providing abstraction mechanisms for modeling, which includes:

- Classes
- Objects

Notes

- Inheritance
- Association

Self Assessment

Fill in the blanks:

4. A ..... is an abstraction of something for the purpose of understanding it before building it.
5. ..... help in understanding problems, communicating with experts from a distance, modeling enterprises, and designing programs and database.
6. The ..... model is a concise, precise abstraction and agreement on how the desired system must be developed.
7. At ..... stage, the complete system architecture is designed.
8. The ..... decides the data structures and algorithms needed to implement each of the classes in the system.

2.3 Abstraction

Abstraction is one of the very important concepts of object oriented systems. Abstraction is a simplified description, or specification, of a system that emphasizes some of the system's details or properties while suppressing others. A good abstraction is one that emphasizes details that are significant to the reader or user and suppresses details that are, at least for the moment, immaterial or diversionary. Abstraction focuses on the essential, inherent aspects of an object of the system. It does not represent the accidental properties of the system. In system development, abstraction helps to focus on what an object is supposed to do, before deciding how it should be implemented. The use of abstraction protects the freedom to make decisions for as long as possible, by avoiding intermediate commitments in problem solving. Most of the modern languages provide data abstraction. With the abstraction, ability to use inheritance and ability to apply polymorphism provides additional freedom and capability for system development.

Did u know? When you are using abstraction during analysis, you have to deal with application-domain concepts. You do not have to design and make implementation decisions at that point.

One point of confusion regarding abstraction is its use as a process and as an entity. Abstraction, as a process, denotes the extracting of the essential details about an item, or a group of items, while ignoring the inessential details. Abstraction, as an entity, denotes a model, a view, or some other focused representation for an actual item. Abstraction is most often used as a complexity mastering technique.

For example, we often hear people say things like "just give the highlights" or "just the facts, please."

What these people are asking for are abstractions. We can have varying degrees of abstraction, although these 'degrees' are more commonly referred to as 'levels'.

As we move to higher levels of abstraction, we focus on the larger and more important pieces of information (using our chosen selection criteria). Another common observation is that as we move to higher levels of abstraction, we tend to concern ourselves with progressively smaller

volumes of information and fewer overall items. As we move to lower levels of abstraction, we reveal more detail, typically encounter more individual items, and increase the volume of information with which we must deal.

We also note that there are many different types of abstraction.

For example, functional abstraction, data abstraction, process abstraction and object abstraction.

Usually, abstraction is not defined in terms of information hiding.

For example, note the use of words such as 'ignore' and 'extracting'. However, we should also note the use of the words 'suppress' and 'suppressing' in some of the examples.

In short one might say that abstraction dictates that some information is more important than other information, but (correctly) does not specify a specific mechanism for handling the unimportant information.

Self Assessment

Fill in the blanks:

9. ..... is a simplified description, or specification, of a system that emphasizes some of the system's details or properties while suppressing others.
10. Abstraction, as an ....., denotes a model, a view, or some other focused representation for an actual item.
11. When you are using abstraction during analysis, you have to deal with ..... concepts.

2.4 Three Models

Whole object oriented modeling is covered by using three kinds of models for a system description. These models are:

- Object model,
- Dynamic model, and
- Functional model

Object models are used for describing the objects in the system and their relationship among each other in the system. The object model encompasses the principles of abstraction, encapsulation, modularity, hierarchy, typing, concurrency, and persistence. The Object-Oriented analysis and design is fundamentally different than traditional structured design approaches; it requires a different method of thinking about decomposition, and it produces software architectures that are largely outside the realm of the structured design culture. These differences arise from the fact that structured design methods build upon structured programming, whereas object-oriented design builds upon object-oriented programming.

The dynamic model describes interaction among objects and information flow in the system. The dynamic model shows the time dependent behavior of the system and the objects in it. Begin dynamic analysis by looking for event, externally visible stimuli and responses.

Notes The dynamic model is important for interactive systems, but insignificant for purely static data repository, such as database.

Notes

The following steps are performed in constructing a dynamic model:

- Prepare scenarios of typical interaction sequences
- Identify events between objects
- Prepare an event trace for each scenario
- Build a state diagram
- Match events between objects to verify consistency

The data transformations in the system are described by a functional model. The functional model shows how values are computed, without regard for sequencing, decisions, or object structure. The functional model shows the values which depend on other values and the functions that relate them with each other.

All three models are applicable during all stages of development. These models bear the responsibility of acquiring implementation details of the system development.

Caution You cannot describe a system completely until unless all three models are described properly.

Task Compare and contrast dynamic model and functional model.

Self Assessment

Fill in the blanks:

12. ..... models are used for describing the objects in the system and their relationship among each other in the system.
13. The ..... model describes interaction among objects and information flow in the system.
14. The ..... model shows how values are computed, without regard for sequencing, decisions, or object structure.
15. The dynamic model is important for interactive systems, but insignificant for purely ..... data repository.

Case Study

Pulsing Together

The healthcare industry has yet to fix data standards. It is more a question of standards for various uses than that of an overlap. CORBAmed (an OMG working group) has issued a Patient Identification Standard (PIDS) that can be used to serve demographic information. Demographic information relates to the patient's name, social security number, medical record number, insurance policy number, birth date, sex and chart details.

CORBAmed is also trying to provide access to clinical data through a Request for Proposal (RFP) on Clinical Observations Access Service, as well as an RFP outstanding that seeks

Contd...

Notes

proposals to manage access control. Another RFP is in the development stage to manage the transcription process.

CORBAmed is based on the Common Object Request Broker Architecture (CORBA) distributed object model. It provides scalable infrastructure for a distributed object model of computing. The CORBAmed domain task force has to develop standards for the "business objects" for healthcare.

By viewing a set of hosts as a network of distributed, communicating "objects", clients are able to obtain processed information over an enterprise-scale network. In the long run, hosts may use each other's distributed object services to provide information on demand, thus providing an alternative to the exchange of HL7 messages. However, CORBAmed will best be used in client-server communications. By providing an object-oriented view of clinical information, CORBAmed allows hosts to "protect" their data by exposing only specific "methods" to clients.

Clients can rely on CORBA hosts to provide information on demand and need not keep duplicates of information in local storage, eliminating problems of freshness and authenticity which are inherent in message exchanges. CORBA standards are platform and language neutral, which is a key requirement for any large-scale client-server integration effort.

HL7 is a messaging standard which is used to exchange healthcare related information via a coded textual message. The messages may be sent ad hoc, e.g., based on certain trigger events in the sending system, or may be sent in response to a query (however, the latter mode is not well supported at present). HL7 has been criticised for the lack of an information model to standardise the semantics of the information being exchanged. This has been rectified in version 3.0.

In addition, real-world HL7 experience will be used to define clinical observation information to be exchanged via the forthcoming CORBAmed Clinical Observation Access Service (COAS). The COAS, request for proposal (RFP) mandates that existing standards (including HL7, DICOM, MIB and others) be looked at for definitions of the information to be exchanged. Further, CORBAmed has an active working group looking at inter-operability with HL7.

HTML/XML: XML stands for extensible markup language. It is a universal language in a network's middleware layer, the conduit that ties together distributed applications with software components based on CORBA (Component Object Request Broker Architecture) COM (Common Object Model) and Enterprise Java.

The healthcare industry has yet to resolve the problem of data confusion. Applications built for legacy mainframe, Unix, NT or Linux systems all define data in patient records, financial statements and insurance forms differently. As expansions and mergers reshuffle IT infrastructures of large healthcare organisations; there must be a common ground for defining data so that old and new applications can communicate with each other.

Universal Pipeline

Healthcare-savvy application developers and integrators find XML particularly appealing. Healthcare providers could eventually use similar Web-based systems for insurance eligibility checks, claims processing and clinical referrals. The tools that were used before XML came in to vogue were data-interchange standards such as Network File System (NFS) or ANSI X.12.

XML's data definitions are simpler and more universal than existing specifications such as Standard Generalised Mark-up Language (SGML) or the proprietary specifications used in

Contd...

Notes

Electronic Data Interchange (EDI) networks. Thus, developers and systems integrators have a simple guidebook for, say, labeling "penicillin" as an adverse-reaction drug in the field "allergies," so patient records in a hospital database and the Web browser in a pharmacy both know where to store the data.

XML also works with existing Web standards. Hypertext Markup Language (HTML) determines how data is displayed on screen. The data passes from the Web to a variety of hardware that connects to the network via TCP/IP. XML labels incoming data so any application knows how to handle it. Because XML can exchange data among different computing platforms it helps solve the inter-operability problems of middleware. Getting various back-end systems that use CORBA or COM components to work together.

The component technologies described in this article are being standardised through various organisations in the US.

Conclusion

Thus, many "flavours" or interpretations of the standard exist today. The problems with standards are that there are so many! This has led many healthcare IT professionals to view standards with some skepticism. However, there is not a great deal of overlap among these standards, and development tries to make use of the best features of existing standards, rather than compete with them.

While it is true that the many standards activities within the healthcare arena are not perfectly aligned, they are generally not competitive. Healthcare IT professionals are not faced with a choice of standards, but a set of standards to use in different circumstances. HL7 will continue to dominate in the host world, although CORBA could play a role, particularly where the retention of data ownership rights is an issue. CORBAmed should provide the standards for client-host computing, as CORBA middleware is robust and proven in enterprise-scale applications.

DICOM should provide the model for observational reporting as well as the representation for medical image data, particularly for primary diagnosis; ASTM on healthcare applications standards; ASC X12 on billing standards and various organisations on coding and vocabulary standards.

Users should insist that they continue to cooperate toward greater reuse and inter-operability.

Healthcare IT can be run by those who have gained experience and success in non-healthcare IT, where IT experts learned to use IT to support organisational goals. The future path to management advancement in healthcare may be through enterprise IT.

Question

Discuss the use of CORBAmed in healthcare industry.

Source: http://www.thehindubusinessline.in/2000/10/25/stories/242539yc.htm

2.5 Summary

- Object-oriented Analysis and Design (OOAD) is a software engineering approach that models a system as a group of interacting objects.
- Object-oriented Analysis (OOA) looks at the problem domain, with the aim of producing a conceptual model of the information that exists in the area being analyzed.
- Modeling is used frequently, during many of the phases of the software life cycle such as analysis, design and implementation.

Notes

- Object oriented models help in understanding problems, communicating with experts from a distance, modeling enterprises, and designing programs and database.
- Abstraction is a simplified description, or specification, of a system that emphasizes some of the system's details or properties while suppressing others.
- Object models are used for describing the objects in the system and their relationship among each other in the system.
- The dynamic model describes interaction among objects and information flow in the system.
- The functional model shows how values are computed, without regard for sequencing, decisions, or object structure.

2.6 Keywords

Abstraction: Abstraction is a simplified description, or specification, of a system that emphasizes some of the system's details or properties while suppressing others.

Dynamic model: The dynamic model describes interaction among objects and information flow in the system.

Function model: The functional model shows how values are computed, without regard for sequencing, decisions, or object structure.

Model: A model is an abstraction of something for the purpose of understanding it before building it.

Object models: Object models are used for describing the objects in the system and their relationship among each other in the system.

Object-oriented modelling (OOM): Object-oriented modelling is all about visualizing the things by using models organized around real world concepts.

Object-oriented Analysis (OOA): Object-oriented Analysis looks at the problem domain, with the aim of producing a conceptual model of the information that exists in the area being analyzed.

Object-oriented Design (OOD): Object-oriented Design transforms the conceptual model produced in object-oriented analysis.

2.7 Review Questions

1. Discuss the concept of object-oriented analysis and design.
2. Examine the situations in which object-oriented modelling is useful.
3. What is modelling? Discuss main reasons for constructing models.
4. Elucidate the concept of object-oriented programming.
5. "With the increasing complexity of systems, importance of modelling techniques increases." Comment.
6. Discuss the various benefits of object-oriented modelling.
7. Describe the use of abstraction in object-oriented systems.
8. What are the different types of abstraction? Enlighten.
9. Analyze the three models included in object-oriented modelling.
10. Make distinction between object model and dynamic model.

Notes

Answers: Self Assessment

1. Object-oriented Analysis (OOA)
2. Object-oriented Design (OOD)
3. analysis
4. Model
5. Object oriented models
6. Analysis
7. system design
8. object design
9. Abstraction
10. Entity
11. application-domain
12. Object
13. dynamic
14. Functional
15. static

2.8 Further Readings

Books

Booch, Grady, (1994), *Object-oriented Analysis & Design,* Addison Wesley

Laganiere, Robert, (2004), *Object-oriented Software Engineering,* TMH

Rumbaugh, J. (2007), *Object-oriented Modelling and Design with UML*, Pearson Education

Satzinger, (2007), *Object-oriented Analysis & Design with the Unified Process*, Thomson

Online links

http://sadiploma.blogspot.in/2008/04/static-modelling-dynamic-modelling-in.html

http://www.cs.sfu.ca/CourseCentral/354/zaiane/material/notes/Chapter1/node8.html

http://www.cse.iitb.ac.in/~rkj/talks/modeling.pdf

http://www.imse.hku.hk/imse1013/pdf/ESA-06%20OOAD.pdf

Notes

Unit 3: Class Modelling

CONTENTS

Objectives

Introduction

3.1 Object and Class Concepts

3.1.1 Class

3.1.2 Objects

3.2 Link and Association Concepts

3.2.1 General Concepts

3.2.2 Multiplicity

3.3 Generalization and Inheritance

3.3.1 Generalization

3.3.2 Inheritance

3.4 Sample Class Model

3.4.1 Navigation of Class Model

3.5 Summary

3.6 Keywords

3.7 Review Questions

3.8 Further Readings

Objectives

After studying this unit, you will be able to:

- Discuss object and class concepts
- Explain link and association concepts
- Discuss generalization and inheritance
- Explain sample class model

Introduction

In this unit, we will discuss various characteristics of class modeling. Before we discuss the characteristics of object oriented modeling, let us see how object oriented development is different from structured development of the system. In the structured approach, the main emphasis is on specifying and decomposing system functionality. Structured approach is seen as the most direct way of implementing a desired goal. A structured approach has certain basic problems, such as, if the requirements of system change then a system based on decomposing functionality may require massive restructuring, and, the system gradually become unmanageable. In contrast to the structured approach, the basic focus of object-oriented approach is to identify objects from the application domain, and then to associate procedures (methods) around these identified

Notes

objects. You can say that object oriented development is an indirect way of system development because in this approach a holistic view of application domain is considered, and objects are identified in the related problem domain. A historic view of application helps in realizing the situations and characteristics of the system. Taking a holistic view of the problem domain rather than considering functional requirements of a single problem give an edge to object oriented development. Once the objects are created with the needed characteristics, they communicate with each other by message passing during problem solving.

3.1 Object and Class Concepts

3.1.1 Class

A class is a collection of things, or concepts that have the same characteristics. Each of these things or concepts is called an object. Classes define the basic words of the system being modeled. A class describes a collection of similar objects. It is a template where certain basic characteristics of a set of objects are defined. A class defines the basic attributes and the operations of the objects of that type. Using a set of classes as the core vocabulary of a software project tends to greatly facilitate understanding and agreement about the meanings of terms, and other characteristics of the objects in the system.

Classes can serve as the foundation for data modeling. In OOM, the term classes is usually the base from which visual modeling tools such as Rational Rose XDE, Visual Paradigm function and design the model of systems.

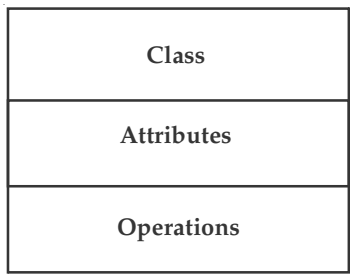
Defining a class does not define any object, but it only creates a template. For objects to be actually created, instances of the class are to be created as per the requirement of the case. Now, let us see how the characteristics that classes are captured as attributes and operations. These terms are defined as follows:

- Attributes are named slots for data values that belong to the class
- Operations represent services that an object can request to affect the behaviour of the object or the system itself.

Did u know? Different objects of a given class typically have at least some differences in the values of their attributes.

The notation for a class is a box with three sections. The top section contains the name of the class in boldface type, the middle section contains the attributes that belong to the class, and the bottom section contains the class's operations as you can see in Figure 3.1.

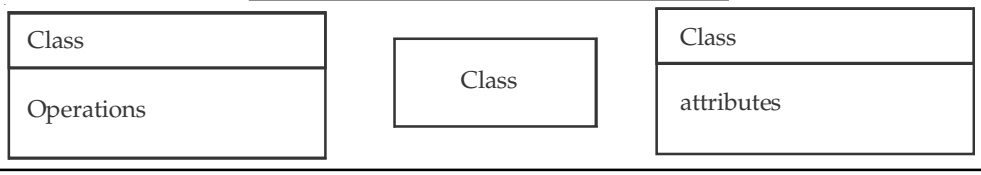
Figure 3.1: Class notation



Notes

You can, also show a class without its attributes or its operations, or the name of the class can appear by itself as shown in Figure 3.2.

Figure 3.2: Alternate Class Notations



The naming conventions for classes are as follow:

- Class names are simple nouns or noun phrases.
- Attribute names in a class are simple nouns or noun phrases. The first word is not capitalized, but subsequent words may be capital.
- Operation names are simple verbs. As with attributes, the first word is not capitalized and subsequent words may be capital.

3.1.2 Objects

The notation for an object is the same in basic form as that for a class. There are three differences between the notations, which are:

- Within the top section of the class box, the name of the class to which the object belongs appears after a colon. The object may have a name, which appears before the colon, or it may be anonymous, in which case nothing appears before the colon.
- The contents of the top compartment are underlined for an object.
- Each attribute defined for the given class has a specific value for each object that belongs to that class.

You can see the notion of an object you can see in Figure 3.3.

Figure 3.3: Notation of Object

Example: If you look around you will find many examples of real world objects such as your books, your desk, your television, etc.

Notes Everything that the software object knows (state) and can do (behaviour) is expressed by the variables and the methods within that object. In other words, all the objects share states and behaviour. Let us say that a software object that models your real world bicycle would have variables that indicated the bicycle's current state: its speed is 20 mph, and its current gear is the 3rd gear, etc.

Notes

Communication by Message Passing

You will agree that a single object alone is generally not very useful. Objects usually appear as a component of a larger program or a system. Through the interaction of these objects, functionality of systems is achieved. Software objects interact and communicate with each other by message passing to each other. When object X wants object Y to perform one of methods of object Y, object X sends a message to object Y. Message passing provide two significant benefits:

- An object's characteristics are expressed through its methods, so message passing supports all possible interactions between objects.
- It closes the gap between objects. Objects do not need to be in the same process, or even on the same machine, to send and receive messages back and forth to each other.

Task Illustrate attributes and operations with example.

Self Assessment

Fill in the blanks:

1. A class describes a collection of similar .....
2. ..... are named slots for data values that belong to the class.
3. ..... represent services that an object can request to affect the behavior of the object or the system itself.
4. The notation for an object is the same in basic form as that for a .....
5. Software objects interact and communicate with each other by ..... to each other.

3.2 Link and Association Concepts

Links and associations are the basic means used for establishing relationships among objects and classes of the system.

3.2.1 General Concepts

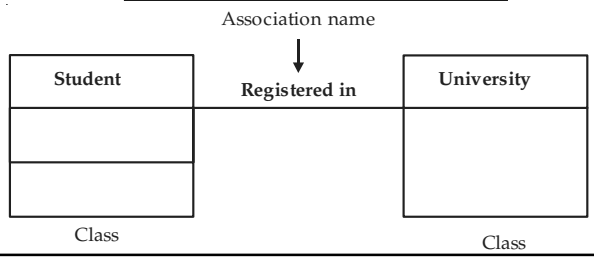
A link is a physical or conceptual connection between objects, for example, a student, Ravi study in IGNOU. Mathematically, you can define a link as a tuple that is an ordered list of objects. Further, a link is also defined as an instance of an association. In other words you can say that an association is a group of links with a common structure and common meanings, for example, a student study in a university. All the links in an association connects objects from the same classes. A link is used to show a relationship between two (or more) objects.

Association and classes are similar in the sense that classes describe objects, and association describes links.

Example: In Figure 3.4 shows us how we can show the association between Student and University.

Notes

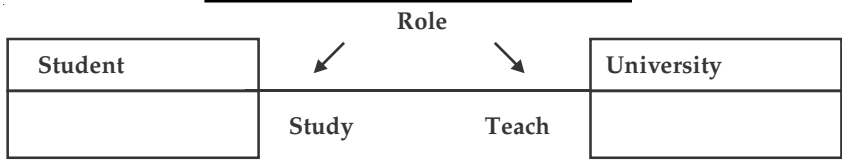
Figure 3.4: Association



Caution Every association has roles.

For example, in Figure 3.5 you can see that two classes, Student and University, have their defined roles. Here you can also see that binary association has two roles, one from each class.

Figure 3.5: Roles in Association

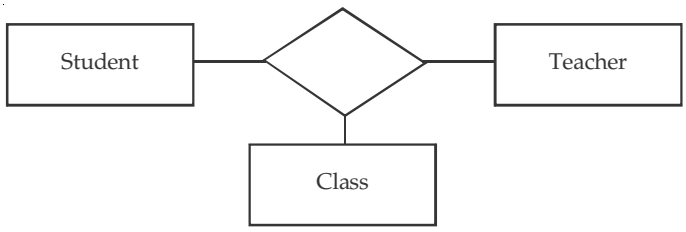


Associations may be binary, ternary, or have higher order. In exercise, the vast majority of association is binary or ternary associations. But a ternary association is formed compulsion; they cannot be converted into binary association.

Did u know? If a ternary association is decomposed in some other association, some information will be lost.

Example: In Figure 3.6 you can see a ternary association.

Figure 3.6: Ternary Association



Task Make distinction between binary and ternary association.

3.2.2 Multiplicity

Multiplicity in an association specifies how many objects participate in a relationship. Multiplicity decides the number of related objects. Multiplicity is generally explained as "one" or "many," but in general it is a subset of the non-negative integers.

Notes

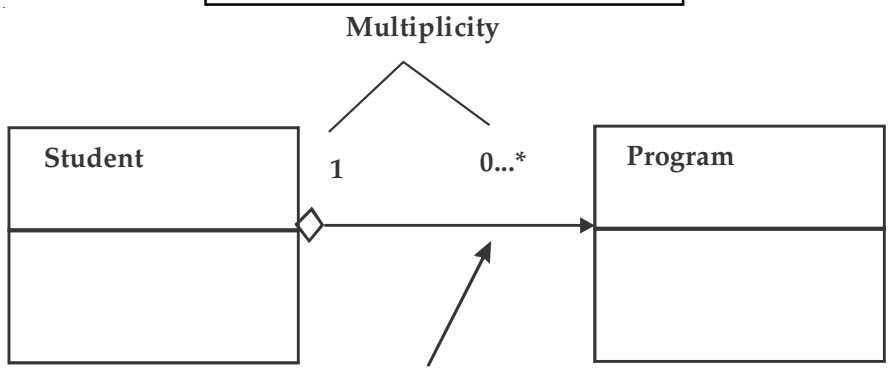
Table 3.1: Multiplicity Indicators

Indicator	Meaning
0..1	Zero or one
1	One only
0..*	Zero or more
1..*	One or more
n	Only *n* (where *n*> 1)
0..n	Zero to *n* (where *n*> 1)
1..n	One to *n* (where *n*> 1)

In associations, generally movement is in both the directions of the relationships but if you want to be specific in any particular direction, you have to mark it by an arrow.

Example: An example of multiplicity is shown in Figure 3.7.

Figure 3.7: Association and Movement



Self Assessment

Fill in the blanks:

6. A ..... is a physical or conceptual connection between objects.
7. A link is also defined as an instance of an .....
8. If a ..... association is decomposed in some other association, some information will be lost.
9. ..... in an association specifies how many objects participate in a relationship.

3.3 Generalization and Inheritance

In this section, we will discuss the concepts of generalization, inheritance, and their uses in OOM.

3.3.1 Generalization

Generalization and inheritance are powerful abstractions for sharing the structure and/or behaviour of one or more classes.

Notes

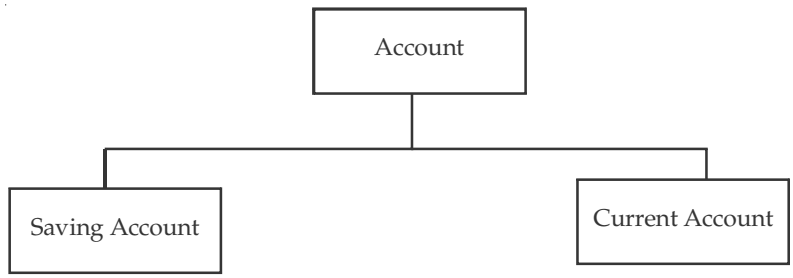
Generalization is the relationship between a class, and it defines a hierarchy of abstraction in which subclasses (one or more) inherit from one or more superclasses. Generalization and inheritance are transitive across a subjective number of levels in the hierarchy. Generalization is an "is-a-kind of" relationship. For instance, Saving Account is a kind of Account; PG student is kind of Student, etc.

The notation for generalization is a triangle connecting a super class to its subclasses. The superclass is connected by a line to the top of the triangle. The subclasses are connected by lines to a horizontal bar attached to the base of the triangle.

Caution Generalization is a very useful construct for both abstract modeling and implementation.

Example: A generalization of account class is shown in Figure 3.8.

Figure 3.8: Generalization of Account Class



Source: http://vedyadhara.ignou.ac.in/wiki/images/a/aa/B1U1mcs-032.pdf

3.3.2 Inheritance

Inheritance is taken in the sense of code reuse within the object oriented development. During modeling, we look at the resulting classes, and try to group similar classes together so that code reuse can be enforced. Generalization, specialization, and inheritance have very close association. Generalization is used to refer to the relationship among classes, and inheritance is used for sharing attributes and operations using the generalization relationship.

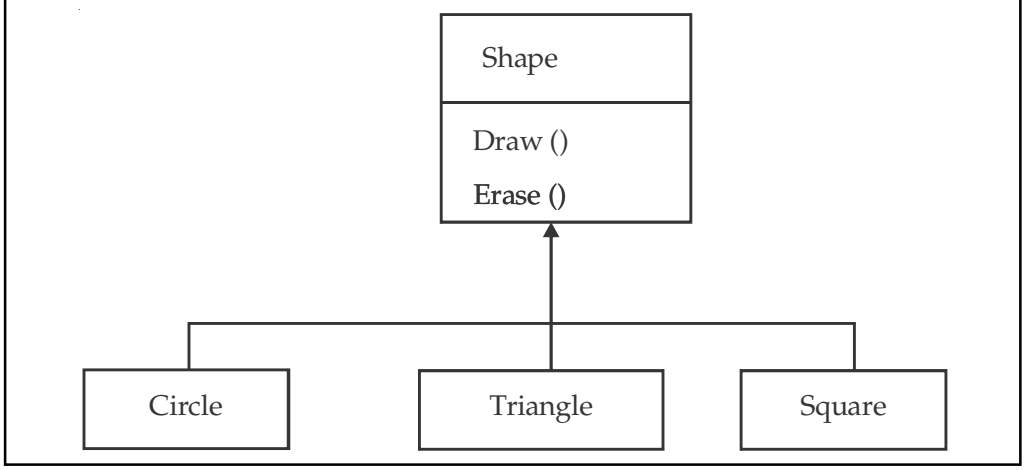
Notes In respect of inheritance, generalization and specialization are two phases of a coin in the sense that if a subclass is seen from a superclass the subclass is seen as a specialized version of superclass and in, reverse, a superclass looks like general form of subclass.

During inheritance, a subclass may override a superclass feature by defining that feature with the same name. The overriding features (the subclass feature with the same names of superclass features) refine and replace the overridden feature (the superclass feature).

Example: Let us look at the diagram given in Figure 3.9. In this diagram, Circle, Triangle, and Square classes are inherited from Shape class. This is a case of single inheritance because here, one class inherits from only one class.

Notes

Figure 3.9: Single Inheritance

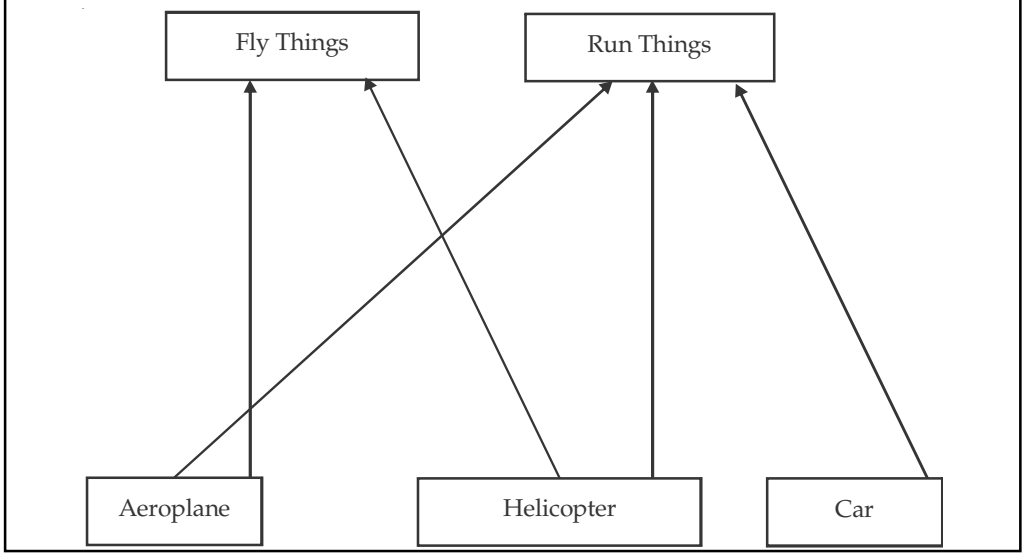


Source: http://vedyadhara.ignou.ac.in/wiki/images/a/aa/B1U1mcs-032.pdf

In multiple inheritances, one class is inherited from more than one class.

Example: Multiple inheritances is shown in Figure 3.10.

Figure 3.10: Multiple Inheritance



Source: http://vedyadhara.ignou.ac.in/wiki/images/a/aa/B1U1mcs-032.pdf

Self Assessment

Fill in the blanks:

10. The notation for ..... is a triangle connecting a super class to its subclasses.
11. ..... is used for sharing attributes and operations using the generalization relationship.
12. In ..... inheritance, one class is inherited from more than one class.

Notes

13. During inheritance, a subclass may override a ..... feature by defining that feature with the same name.

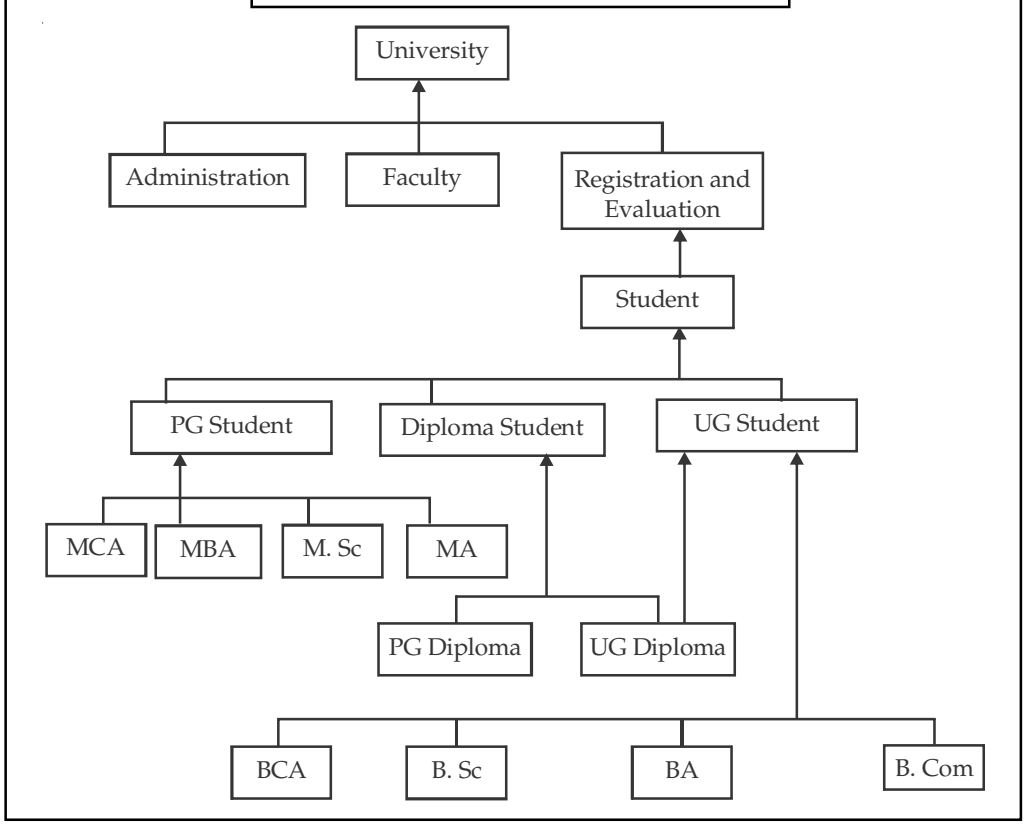
3.4 Sample Class Model

In object oriented modeling, we understand the system and on the basis of that classes are identified. Establishing relationship among different classes in the system is the first and foremost activity. Here, we have a simple model of a University System with respect to different levels of courses offered by the University. As you can see in Figure 3.11, we have given the basic classes of this system.

3.4.1 Navigation of Class Model

This diagram covers different levels of students in the hierarchy. Similarly, for other classes, such as Administration and Faculty, hierarchy level can be drawn to give a broader view of whole system.

Figure 3.11: Object Model for University System



Self Assessment

Fill in the blanks:

14. In ....., we understand the system and on the basis of that classes are identified.

15. Establishing ..... among different classes in the system are the first and foremost activity.

Notes

Case Study

A Payroll Program

Consider a payroll program that processes employee records at a small manufacturing firm. This company has three types of employees:

- ***Managers:*** Receive a regular salary.
- ***Office Workers:*** Receive an hourly wage and are eligible for overtime after 40 hours.
- ***Production Workers:*** Are paid according to a piece rate.

Structured Approach

```
FOR EVERY EMPLOYEE DO
BEGIN
IF employee = manager THEN
CALL computeManagerSalary
IF employee = office worker THEN CALL computeOfficeWorkerSalary
IF employee = production worker THEN CALL computeProductionWorkerSalary
END
```

What if we add two new types of employees?

Temporary office workers ineligible for overtime, junior production workers who receive an hourly wage plus a lower piece rate.

```
FOR EVERY EMPLOYEE DO
BEGIN
IF employee = manager THEN
CALL computeManagerSalary
IF employee = office worker THEN
CALL computeOfficeWorker_salary
IF employee = production worker THEN
CALL computeProductionWorker_salary
IF employee = temporary office worker THEN
CALL computeTemporaryOfficeWorkerSalary
IF employee = junior production worker THEN
CALL computeJuniorProductionWorkerSalary
END
```

Contd...

Notes

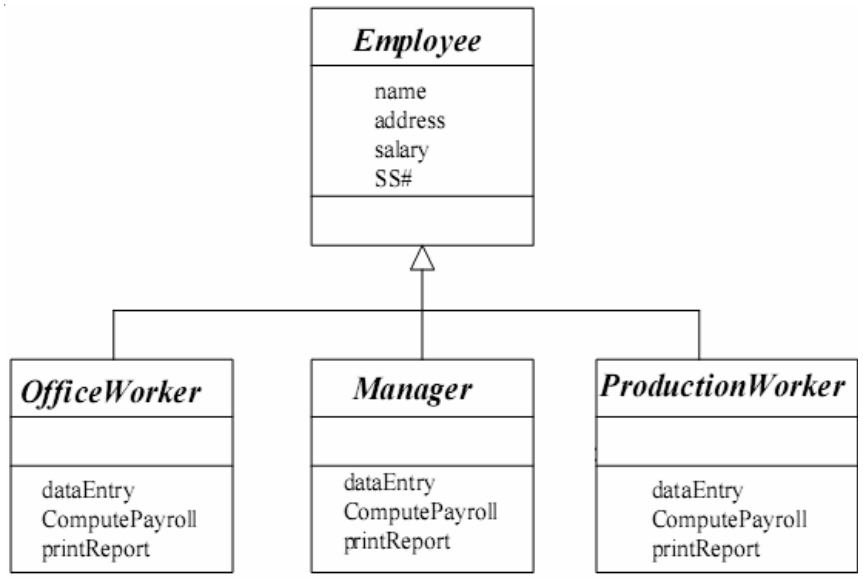
An Object-oriented Approach

The goal of OO analysis is to identify objects and classes that support the problem domain and system's requirements.

Some general candidate classes are:

- Persons
- Places
- Things
- Class Hierarchy
- Identify class hierarchy
- Identify commonality among the classes
- Draw the general-specific class hierarchy.

Figure 1: Class Hierarchy



OO Approach

FOR EVERY EMPLOYEE DO

BEGIN

employee computePayroll

END

Question

What objects does the application need? Discuss.

Source: http://www.einsteincollege.ac.in/Assets/Department/Lecturer%20notes/CSE/UG/CS%201042%20OBJECT%20ORIENTED%20ANALYSIS%20AND%20DESIGN.pdf

Notes

3.5 Summary

- A class describes a collection of similar objects. It is a template where certain basic characteristics of a set of objects are defined.
- The notation for an object is the same in basic form as that for a class.
- Objects usually appear as components of a larger program or a system. Through the interaction of these objects, functionality of systems is achieved.
- Links and associations are the basic means used for establishing relationships among objects and classes of the system.
- Association and classes are similar in the sense that classes describe objects, and association describes links.
- Multiplicity in an association specifies how many objects participate in a relationship. Multiplicity decides the number of related objects.
- Generalization is the relationship between a class, and it defines a hierarchy of abstraction in which subclasses (one or more) inherit from one or more superclasses.
- Generalization is used to refer to the relationship among classes, and inheritance is used for sharing attributes and operations using the generalization relationship.

3.6 Keywords

Association: Association defines a relationship between two or more classes that specifies connections among their instances.

Attributes: Attributes are named slots for data values that belong to the class.

Class: Class is a template where certain basic characteristics of a set of objects are defined.

Generalization: Generalization is the relationship between a class, and it defines a hierarchy of abstraction in which subclasses (one or more) inherit from one or more superclasses.

Inheritance: Inheritance is a relationship between classes where one class is the parent class of another (derived) class.

Link: A link is a physical or conceptual connection between objects.

Multiplicity: Multiplicity in an association specifies how many objects participate in a relationship.

Object: An object is anything, real or abstract, about which we store data and those methods that manipulate the data.

3.7 Review Questions

1. What is a class? Make distinction between attributes and operations.
2. Describe the differences between the notations of class and object respectively.
3. Illustrate the concept of objects and classes with examples.
4. Explain the concept of communication by message passing. Also discuss the advantages of message passing.
5. Elucidate the concept of link and association with example.

Notes

6. What does multiplicity in an association specify? Illustrate.
7. "Generalization and inheritance are transitive across a subjective number of levels in the hierarchy." Comment.
8. Describe the uses of Generalization in object-oriented modelling.
9. What is multiple inheritance? Illustrate with example.
10. Take any example and show the relationship among different classes in the system.

Answers: Self Assessment

1.	Objects	2.	Attributes
3.	Operations	4.	Class
5.	message passing	6.	Link
7.	association	8.	ternary
9.	Multiplicity	10.	Generalization
11.	Inheritance	12.	Multiple
13.	superclass	14.	object oriented modeling
15.	relationship		

3.8 Further Readings

Books

Booch, Grady, (1994), *Object-oriented Analysis & Design,* Addison Wesley

Laganiere, Robert, (2004), *Object-oriented Software Engineering*, TMH

Rumbaugh, J. (2007), *Object-oriented Modelling and Design with UML,* Pearson Education

Satzinger, (2007), *Object-oriented Analysis & Design with the Unified Process,* Thomson

Online links

http://www.enel.ucalgary.ca/People/far/Lectures/SENG401/PDF/OOAD_with_UML.pdf

http://www.mppmu.mpg.de/english/kluth_oo_uml.pdf

http://www.slideshare.net/anniyappa/ooad-overview

http://www.trainingetc.com/PDF/TE1802eval.pdf

Notes

Unit 4: Advanced Class Modelling – I

CONTENTS

Objectives

Introduction

4.1 Advanced Object and Class Concepts

4.1.1 Instantiation

4.1.2 Class Attributes and Operations

4.1.3 Attribute Multiplicity

4.1.4 Candidate Keys for Classes

4.1.5 Domains

4.2 Association Ends

4.2.1 Rolename

4.2.2 Navigation

4.2.3 Qualifiers

4.3 N-ary Association

4.4 Aggregation

4.5 Summary

4.6 Keywords

4.7 Review Questions

4.8 Further Readings

Objectives

After studying this unit, you will be able to:

- Describe advanced object and class concepts
- Explain the concept of association ends
- Discuss the concept of N-ary associations
- Analyse the concept of aggregation
- Discuss the different types of aggregation

Introduction

Advanced class modeling refers to the advanced properties of objects in general in a specific computer programming language, technology, notation or methodology that uses them. This unit explains the advanced aspects of object modeling that you will need to model complex and large applications. It builds on the basic concepts of the previous unit, so you should study the previous unit properly before going through this unit. This unit will explain the concepts of

Notes

advanced objects and classes, association ends, n-ary association. An object is a something (a thing or a concept) that has a well-defined role in the application domain. Advanced object and class concepts include instantiation, Class Attributes and Operations, Attribute Multiplicity which specifies the possible number of values for an attribute, Candidate Keys for Classes which uniquely identifies the object, domains, etc. Also we will discuss the concept of n-ary association and aggregation.

4.1 Advanced Object and Class Concepts

Various concepts of advanced objects and classes are discussed below:

4.1.1 Instantiation

Instantiation is the relationship between an object and its class. The notation for instantiation is a dashed line from the instance to the class with an arrow pointing to the class; the dashed line is labeled with the legend instance enclosed by guillemets («»). Figure 4.1 shows this notation for City and its two instances Bombay and Prague. Making the instantiation relationship between classes and instances explicit in this way can be helpful in modeling complex problems and in giving examples.

Figure 4.1: Instantiation Relationships

4.1.2 Class Attributes and Operations

A class attribute is an attribute whose value is common to a group of objects in a class rather than peculiar to each instance. Class attributes can be used to store default or summary data for objects. A class operation is an operation on a class rather than on instances of the class. The most common kind of class operations are operations to create new class instances. You can denote class attributes and class operations with an underline. Our convention is to list them at the top of the attribute box and operation box, respectively.

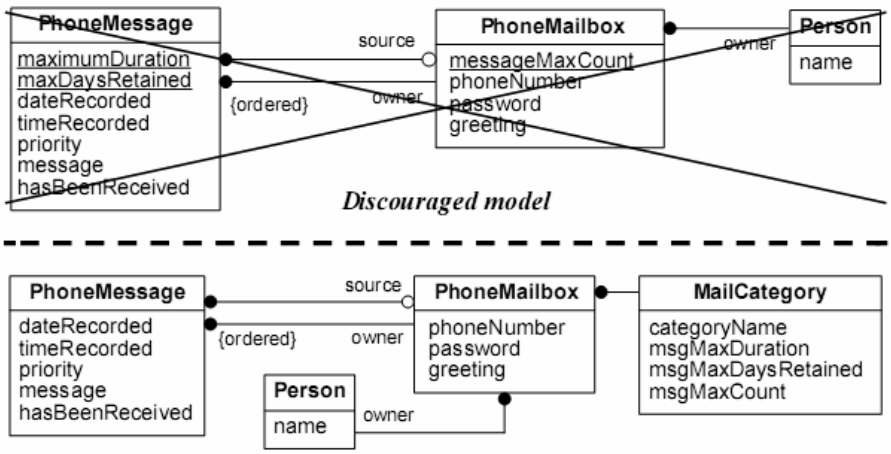
In most applications class attributes can lead to an inferior model. We discourage the use of class attributes. Often you can improve your model by explicitly modeling groups and specifying scope.

Example: The upper model in Figure 4.2 shows class attributes for a simple model of phone mail. Each message has an owner mailbox, date recorded, time recorded, priority, and message contents, indicating if it has been received. A message may have a mailbox as the source or it may be from an external call. Each mailbox has a phone number, password, and recorded greeting. For the PhoneMessage class we can store the maximum duration for a message and the maximum days a message will be retained. For the PhoneMailbox class we can store the maximum number of messages that can be stored.

The upper model is inferior, however, because the maximum duration, maximum days retained, and maximum message count has a single value for the entire phone mail system. In the lower model these limits can vary for different kinds of users, yielding a phone mail system that is more flexible and extensible.

Notes

Figure 4.2: Instead of using Class Attributes, Model Groups Explicitly

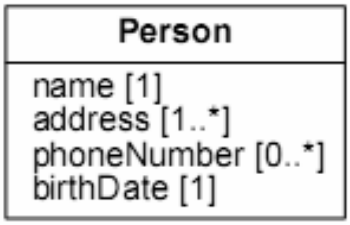


4.1.3 Attribute Multiplicity

Attribute multiplicity specifies the possible number of values for an attribute and is listed in brackets after the attribute name. You may specify a mandatory single value [1], an optional single value [0..1], an unbounded collection with a lower limit [lowerLimit..*], or a collection with xed limits [lower limit..upper limit]. A lower limit of zero allows null values; a lower limit of one or more forbids null values.

If you omit attribute multiplicity, an attribute is assumed to be single valued with nullability unspecified ([0..1] or [1]). In Figure 4.3 a person has one name, one or more addresses, zero or more phone numbers, and one birth date. Attribute multiplicity is similar to multiplicity for associations.

Figure 4.3: Attribute Multiplicity



Did u know? Null is a special value denoting that an attribute value is unknown or not applicable.

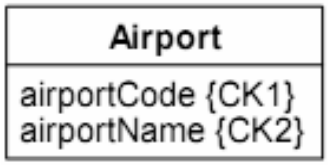
4.1.4 Candidate Keys for Classes

A candidate key for a class is a combination of one or more attributes that uniquely identifies objects within a class. The collection of attributes in a candidate key must be minimal; no attribute can be discarded from the candidate key without destroying uniqueness. No attribute in a candidate key can be null. A given attribute may participate in multiple candidate keys.

Notes

Example: In Figure 4.4 airportCode and airportName are two candidate keys for Airport. The model species that each airportCode (such as IAH, HOU, STL, ALB) uniquely identifies an airport. Each airportName (such as Houston Intercontinental, Houston Hobby, Lambert St. Louis airport, and Albany NY airport) also uniquely identifies an airport.

Figure 4.4: Candidate Keys for a Class



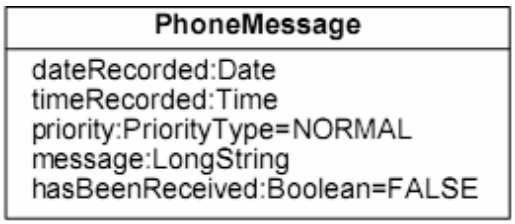
We indicate a candidate key for a class with the notation CKn in braces next to the appropriate attributes. The n is a number that differentiates multiple candidate keys. For a multi-attribute candidate key, multiple attributes have the CKn designation with the same value of n.

Some may recognize the term "candidate key" from the database literature, but the notion of a candidate key is a logical construct, not an implementation construct. It is often helpful to be able to specify the constraint that one or more attributes taken together are unique. Relational database managers and most object-oriented database managers can readily enforce candidate keys.

4.1.5 Domains

A domain is the named set of possible values for an attribute. The notion of a domain is a fundamental concept in relational DBMS theory, but really has broader applicability as a modeling concept. As Figure 4.5 shows, an attribute name may be followed by a domain and default value. The domain is preceded by a colon; the default value is preceded by an equal sign. Some domains are infinite, such as the set of integers; others are finite.

Figure 4.5: Assign a Domain to an Attribute rather than Directly Assign a Data Type



An enumeration domain is a domain that has a finite set of values. The values are often important to users, and you should carefully document them for your object models.

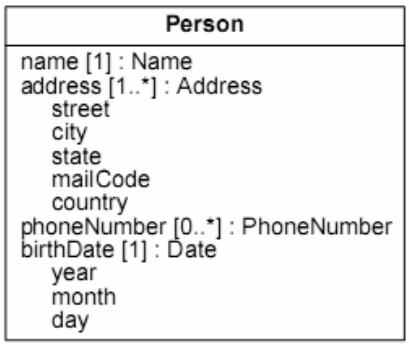
Example: You would most likely implement priority type in Figure 4.5 as an enumeration with values that could include normal, urgent, and informational.

A structured domain is a domain with important internal detail. You can use indentation to show the structure of domains at an arbitrary number of levels.

Example: In Figure 4.6 shows two attributes that have a structured domain. An address consists of a street, city, state, mail code, and country. A birth date has a year, month, and day.

Notes

Figure 4.6: Structured Domains



During analysis you can ignore simple domains, but you should note enumerations and structured domains. During design you should elaborate your object model by assigning a domain to each attribute. During implementation you can then bind each domain to a data type and length.

Domains provide several benefits:

- ***Consistent assignment of data types.*** You can help ensure that attributes have uniform data types by first binding attributes to domains and then binding domains to data types.
- ***Fewer decisions.*** Because domains standardize the choices of data type and length, there are fewer implementation decisions.
- ***Extensibility.*** It is easier to change data types when they are not directly assigned.
- ***Check on validity of operations.*** Finally, you can use the semantic information in domains to check the appropriateness of certain operations.

Example: It may not make sense to compare a name to an address.

Do not confuse a domain with a class. Figure 4.7 summarizes the differences between domains and classes. The objects of a class have identity, may be described by attributes, and may have rich operations. Classes may also be related by associations. In contrast, the values of a domain lack identity.

Example: There can be many Jim Smith objects, but the value normal has only one occurrence.

Most domain values have limited operations and are not described by attributes. During analysis we distinguish between domains and classes according to their semantic intent, even though some domains may be implemented as classes.

Figure 4.7 Classes and Domains differ According to Semantic Intent

Classes	Domains
A class describes objects. • Objects have identity. • Objects may be described by attributes. • Objects may have rich operations. • Classes may be related by associations.	A domain describes values. • Values have no identity. • Most values are not described by attributes. • Most values have limited operations. • Domains do not have associations.

Source: http://www.pearsonhighered.com/assets/hip/us/hip_us_pearsonhighered/samplechapter/0131238299.pdf

Notes

Do not confuse an enumeration domain with generalization. You should introduce generalization only when at least one subclass has significant attributes, operations, or associations that do not apply to the superclass. Do not introduce a generalization just because you have found an enumeration domain.

Notes You can define a domain intensionally (by formula), extensionally (by explicitly listing occurrences), or in terms of another domain.

Task Make distinction between enumeration domain and structured domain.

Self Assessment

Fill in the blanks:

1. ..... is the relationship between an object and its class.
2. A ..... is an operation on a class rather than on instances of the class.
3. Attribute ..... specifies the possible number of values for an attribute and is listed in brackets after the attribute name.
4. The collection of ..... in a candidate key must be minimal.
5. An ..... domain is a domain that has a finite set of values.

4.2 Association Ends

An association end is an endpoint of the line drawn for an association, and it connects the association to a class. An association end may include any of the following items to express more detail about how the class relates to the other class or classes in the association:

- Rolename
- Multiplicity specification
- Aggregation
- Qualifier

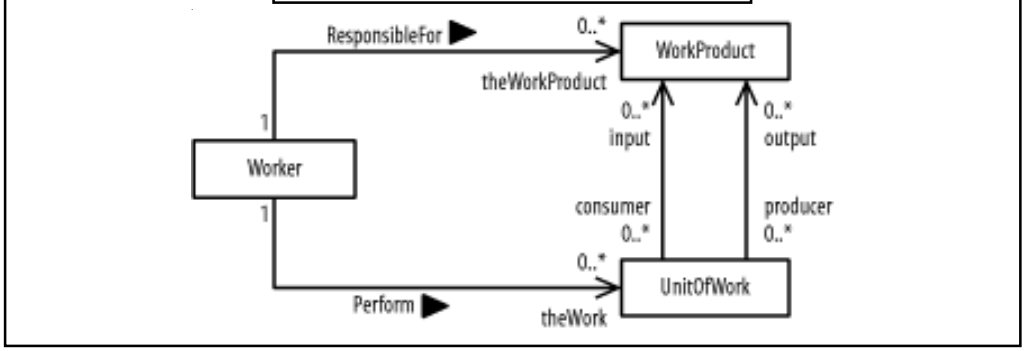
4.2.1 Rolename

A rolename is optional and indicates the role a class plays relative to the other classes in an association, how the other classes "see" the class or what "face" the class projects to the other classes in the relationship. A rolename is shown near the end of an association attached to a class.

Example: A work product is seen as input by a unit of work where the unit of work is seen as a consumer by the work product; a work product is seen as output by a unit of work where the unit of work is seen as a producer by the work product, as shown in Figure 4.8.

Notes

Figure 4.8: Binary Association Ends



Source: http://etutorials.org/Programming/Learning+uml/Part+II+Structural+Modeling/Chapter+3.+Class+and+Object+Diagrams/3.2+Associations+and+Links/

4.2.2 Navigation

Navigation is optional and indicates whether a class may be referenced from the other classes in an association. Navigation is shown as an arrow attached to an association end pointing toward the class in question. If no arrows are present, associations are assumed to be navigable in all directions, and all classes involved in the association may reference one another.

Example: Given a worker, you can determine his work products and units of work. Thus, Figure 4.8 shows arrows pointing towards work product and units of work. Given a unit of work, you can determine its input and output work products; but given a work product, you are unable to identify which worker is responsible for it or which units of work reference it as input or output (as shown in Figure 4.8 by the lack of arrows pointing to the Worker class).

4.2.3 Qualifiers

A qualifier is an attribute of an association class that reduces the multiplicity across an association.

Example: Figure 4.8 shows that multiplicity between work products and units of work is zero or more for both associations; that is, there may be many work products associated with a single unit of work and there may be many units of work associated with a single work product.

Rather than simply say that there are "many" objects involved in the relationship, you can communicate a more finite number.

You can reduce the multiplicity between work products and units of work by asking yourself what you need to know about a unit of work so that you can define a more specific multiplicity one that isn't unbounded on the high-end. Likewise, you can ask yourself the same question about the association between work product and units of work. If you have a work product and the name of a unit of work, you can determine whether a relationship exists between the two; likewise, if you have a unit of work and the name of a work product, you can determine whether a relationship exists between those two. The trick is to document precisely what information is needed so you can identify the objects on the other end of the relationship. This is where the qualifier comes into play.

Essentially, a qualifier is a piece of information used as an index to find the objects on the other end of an association. A qualifier is shown as a small rectangle attached to a class where an object

Notes

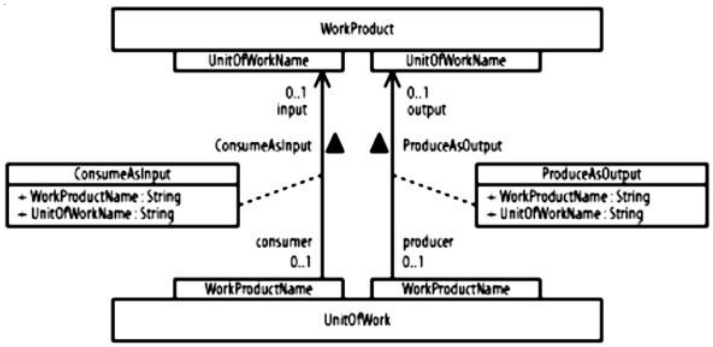
of the class, together with a value for the qualifier, reduces the multiplicity on the other end of the association.

Caution Qualifiers have the same notation as attributes, have no initial values, and must be attributes of the association or the class on the other end of the association.

The relationships between work products and units of work and their qualifiers are shown in Figure 4.9. The qualifiers indicate that a work product with the name of a unit of work may identify a unit of work, and that a unit of work with the name of a work product may identify a work product. This is to be observed that we have reduced the multiplicity of 0..* shown in Figure 4.8 to 0..1 in Figure 4.9. The qualifier enables us to do this.

Did u know? As a UML rule, qualifiers are used only with binary associations.

Figure 4.9: Qualifiers for Associations



Source: http://etutorials.org/Programming/Learning+uml/Part+II+Structural+Modeling/Chapter+3.+Class+and+Object+Diagrams/3.2+Associations+and+Links/

Self Assessment

Fill in the blanks:

6. An ..... is an endpoint of the line drawn for an association, and it connects the association to a class.
7. A ..... is shown near the end of an association attached to a class.
8. ..... is shown as an arrow attached to an association end pointing toward the class in question.
9. A ..... is an attribute of an association class that reduces the multiplicity across an association.
10. A qualifier is a piece of information used as an ..... to find the objects on the other end of an association.

4.3 N-ary Association

The degree of an association is the number of roles for each link. Associations may be binary, ternary, or higher degree. Ternary association and a higher degree association takes place in advanced class modeling.

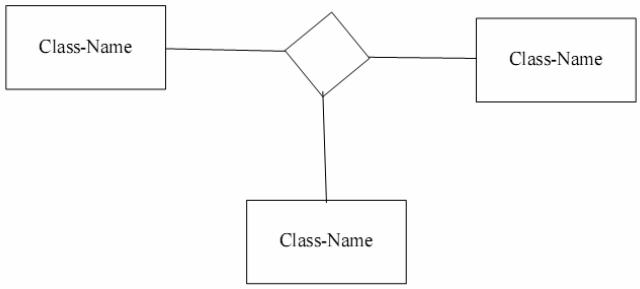
Notes

Ternary associations occasionally occur, but we have rarely encountered an association of higher degree. Ternary association is an association among three classes. On the same line, n-ary association is an association among n classes.

The OMT symbol for ternary and n-ary associations is a diamond with lines connecting to related classes as shown in Figure 4.10.

Caution An n-ary associations cannot be sub-divided into binary associations without losing information.

Figure 4.10: OMT Symbol for Ternary and N-ary Associations

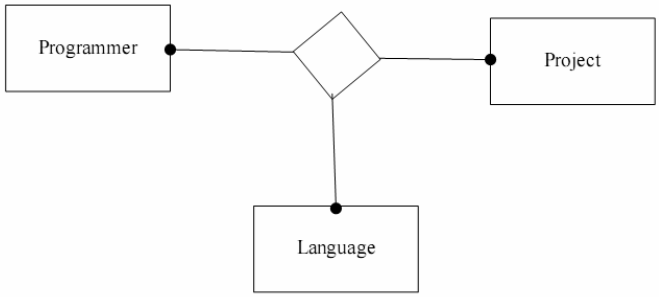


Source: http://www.ddegjust.ac.in/studymaterial/mca-5/mca-503.pdf

Example: Now let us consider the example of a ternary association. Programmers develop Projects in (programming) Languages. One programmer can be engaged in zero, one or more projects and can know zero, one or languages. Similarly, one project can be developed by one or more programmers and in one or more languages. So this association along with its multiplicity is shown in Figure 4.11.

Other examples of ternary and n-ary associations are "Teacher teaches Students in a Classroom", "Doctor diagnoses Patient in Room at a given Schedule" etc.

Figure 4.11: Example of Ternary Association



Source: http://www.ddegjust.ac.in/studymaterial/mca-5/mca-503.pdf

Notes A name for the association is optional and is written next to the diamond.

Self Assessment

Notes

Fill in the blanks:

11. The ..... of an association is the number of roles for each link.

12. ..... association is defined as an association among n classes.

4.4 Aggregation

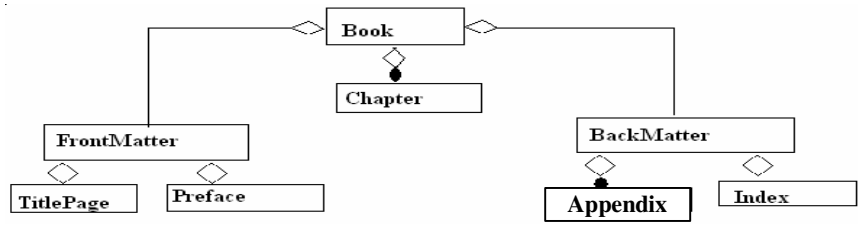
Aggregation is a kind of association, between a whole, called the assembly, and its parts, called the components. Aggregation is often called the "a-part-of" or "parts-explosion" relationship and may be nested to an arbitrary number of levels. Aggregation bears the transitivity property: If A is part of B and B is part of C, then A is part of C.

Aggregation is also antisymmetric: If A is part of B, then B is not part of A. Transitivity lets you compute the transitive closure of an assembly—that is, you can compute the components that directly and indirectly compose it. Transitive closure is a term from graph theory; the transitive closure of a node is the set of nodes that are reachable by some sequence of edges.

Aggregation is drawn like an association with a small diamond added next to the assembly end.

Example: A book consists of front matter, multiple chapters, and back matter. Front matter, in turn, consists of a title page and a preface; back matter consists of multiple appendixes and an index. This is shown in figure 4.12.

Figure 4.12: Aggregation is a Kind of Association with Additional Semantic Properties



Having presented this example, we hasten to add that aggregation does not specify order.

An assembly is a collection of components without any implied order. A book has a well-known and simple order of parts; the back matter follows the chapters which follow the front mater. If the model had to capture component order, you would have to include comments.

An aggregation relationship is essentially a binary association, a pairing between the assembly class and a component class. An assembly with many kinds of components corresponds to many aggregations. We define each individual pairing as an aggregation so that we can specify the multiplicity of each component within the assembly. This definition emphasizes that aggregation is a special form of association. An aggregation can be qualified, have roles, and have link attributes just like any other association.

Aggregation can be fixed, variable or recursive.

- A fixed aggregate has a fixed structure; the number and types of subparts are predefined.
- A variable aggregate has a finite number of levels, but the number of parts may vary.
- A recursive aggregate contains, directly or indirectly, an instance of the same kind of aggregate; the number of potential levels is unlimited.

Notes

Task Illustrate how aggregation is considered as a form of association.

Self Assessment

Fill in the blanks:

13. ..... is a tightly coupled form of association with some extra semantics.
14. In a ..... aggregation, number and subtypes are fixed i.e. predefined.
15. A ..... aggregate contains, directly or indirectly, is an instance of the same aggregate.

4.5 Summary

- Instantiation is the relationship between an object and its class. The notation for instantiation is a dashed line from the instance to the class with an arrow pointing to the class.
- A class attribute is an attribute whose value is common to a group of objects in a class rather than peculiar to each instance.
- Attribute multiplicity specifies the possible number of values for an attribute and is listed in brackets after the attribute name.
- A candidate key for a class is a combination of one or more attributes that uniquely identifies objects within a class.
- An association end is an endpoint of the line drawn for an association, and it connects the association to a class.
- The degree of an association is the number of roles for each link. Associations may be binary, ternary, or higher degree.
- Ternary association is an association among three classes. On the same line, n-ary association is an association among n classes.
- Aggregation is the "part-whole" or "a-part- of" relationship in which objects representing the component of something are associated with an object representing the entire assembly.

4.6 Keywords

Aggregation: Aggregation is a tightly coupled form of association with some extra semantics.

Association end: An association end is an endpoint of the line drawn for an association, and it connects the association to a class.

Candidate key: A candidate key for a class is a combination of one or more attributes that uniquely identifies objects within a class.

Class attribute: A class attribute is an attribute whose value is common to a group of objects in a class rather than peculiar to each instance.

Domain: A domain is the named set of possible values for an attribute.

Instantiation: Instantiation is the relationship between an object and its class.

Multiplicity: Attribute multiplicity specifies the possible number of values for an attribute and is listed in brackets after the attribute name.

Notes

N-ary association: N-ary association is defined as an association among n classes.

Qualifier: A qualifier is an attribute of an association class that reduces the multiplicity across an association.

4.7 Review Questions

1. Explain the concept of class attributes and operations with example.
2. What is attribute multiplicity? Discuss with example.
3. Explain the concept of candidate keys for classes.
4. "It is important to record secondary information without obscuring the focus of an application." Comment.
5. What is an association end? Illustrate with example.
6. Describe the concept of role names and qualifier.
7. Discuss the concept of n-ary association with example.
8. What is aggregation? How is it different from association? Discuss.
9. What are different types of aggregation? Explain each with suitable examples.
10. What is recursive aggregation? Give suitable example of recursive aggregation.

Answers: Self Assessment

1.	Instantiation	2.	class operation
3.	multiplicity	4.	Attributes
5.	enumeration	6.	association end
7.	rolename	8.	Navigation
9.	qualifier	10.	Index
11.	degree	12.	N-ary
13.	Aggregation	14.	Fixed
15.	recursive		

4.8 Further Readings

Books

Booch, Grady, (1994), *Object-oriented Analysis & Design*, Addison Wesley

Laganiere, Robert, (2004), *Object-oriented Software Engineering,* TMH

Rumbaugh, J., (2007), *Object-oriented Modelling and Design with UML,* Pearson Education

Satzinger, (2007), *Object-oriented Analysis & Design with the Unified Process,* Thomson

Online links

http://202.131.97.51/FACULTY/Maitri%20Patel/ooad/solutions%20of%20excercise%20of%20ooad/oo4.pdf

Notes

http://comp.mq.edu.au/books/maciaszek/student_resources/Acro_b&w_3/Ch5_b&w_3.pdf

http://www.is.inf.uni-due.de/courses/mod_ws06/skript/uml-ocl.pdf

http://www.scribd.com/doc/27696643/object-oriented-modeling-and-design

Notes

Unit 5: Advanced Class Modelling – II

CONTENTS

Objectives

Introduction

5.1 Abstract Classes

5.2 Constraints

5.3 Derived Data

5.4 Packages

5.4.1 Logical Horizon

5.5 Summary

5.6 Keywords

5.7 Review Questions

5.8 Further Readings

Objectives

After studying this unit, you will be able to:

- Explain the concept of Abstract Classes
- Discuss Constraints
- Discuss the concept of Derived Data
- Explain Packages

Introduction

In the previous unit, we have discussed advanced object and class concepts, concept of association ends, N-ary associations, concept of aggregation, and different types of aggregation. In this unit, we will discuss some other advanced aspects of object modeling which are required to model complex and large applications. We will discuss the concept of abstract classes, constraints, derived data, and packages. In packages, we will discuss the concept of logical horizon and various examples of packages.

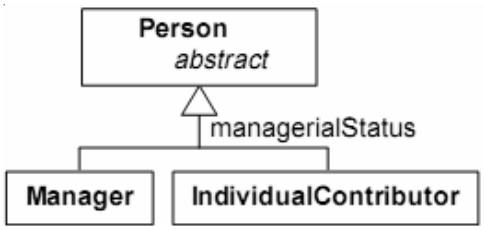
5.1 Abstract Classes

An abstract class is a class that has no direct instances. The descendant classes can also be abstract, but the generalization hierarchy must ultimately terminate in subclasses with direct instances.

Example: In Figure 5.1 Person is an abstract class but the subclasses Manager and Individual Contributor are concrete.

Notes

Figure 5.1: An Abstract Class



Source: http://www.pearsonhighered.com/assets/hip/us/hip_us_pearsonhighered/samplechapter/0131238299.pdf

The legend abstract indicates an abstract superclass. You may define abstract operations for abstract classes. An abstract operation species the signature of an operation, while deferring implementation to the subclasses. The signature of an operation species the argument types, the result type, exception conditions, and the semantics of the operation. The notation for an abstract operation is the legend {abstract} following the operation name.

Self Assessment

Fill in the blanks:

1. An ..... is a class that has no direct instances.
2. An abstract operation species the signature of an operation while deferring implementation to the .....
3. The ..... of an operation species the argument types, the result type, exception conditions, and the semantics of the operation.

5.2 Constraints

A constraint is a functional relationship between modeling constructs such as classes, attributes, and associations. A constraint restricts the values of data. You may place simple constraints in the object model. You should specify complex constraints in the functional model.

A "good" model should capture many constraints with its very structure. In fact, the ability of a model to express important constraints is one measure of the quality of a model. Most object models require several iterations to strike a proper balance between rigor, simplicity, and elegance. However, sometimes it is not practical to express all important constraints with the structure of a model.

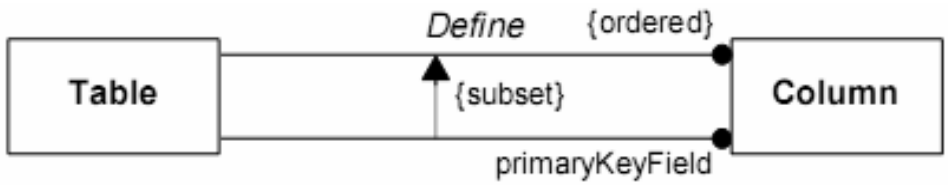
Constraints are denoted by text in braces ("{" and "}"). The text of a constraint should clearly indicate the affected data. Similarly, comments are also delimited by braces. We often use comments to document the rationale for subtle modeling decisions and convey important enumerations.

Did u know? Sometimes it is useful to draw a dotted arrow between classes or associations to indicate the scope of a constraint.

Example: In Figure 5.2 a table has many columns; the primary key columns are a subset of the overall columns.

Notes

Figure 5.2: A Subset Constraint between Associations



Task Why do we draw a dotted arrow between classes or associations? Discuss.

Self Assessment

Fill in the blanks:

4. A ..... is a functional relationship between modeling constructs such as classes, attributes, and associations.
5. A constraint restricts the values of .....
6. The ..... of a constraint should clearly indicate the affected data.
7. We often use ..... to document the rationale for subtle modeling decisions and convey important enumerations.

5.3 Derived Data

Derived data is data that can be completely determined from other data. Classes, attributes, and associations can all be derived. The underlying data can, in turn, be base data or further derived.

Notes Do not confuse our use of the term "derived" with the C++ derived class. A C++ derived class refers to the subclass of a generalization; it has nothing to do with OMT's meaning of derived data.

As a rule, you should not show derived data during analysis unless the data appears in the problem description. During design you can add derived data to improve efficiency and ease implementation. During implementation you can compute derived data on demand from constituent data (lazy evaluation) or precompute and cache it (eager evaluation).

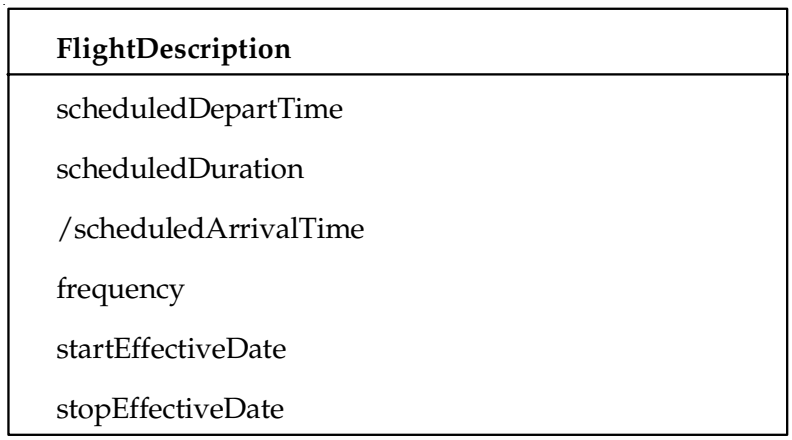
Caution Derived data that is precomputed must be marked as invalid or recomputed if constituent data is changed.

The notation for derived data is a slash preceding the name of the attribute, class, association, or role.

Example: Figure 5.3 shows an example of a derived attribute for airline flightdescriptions.

Notes

Figure 5.3: A Derived Attribute



Self Assessment

Fill in the blanks:

8. ..... is data that can be completely determined from other data.
9. A C++ derived class refers to the subclass of a .....
10. Derived data that is precomputed must be marked as ..... or recomputed if constituent data is changed.

5.4 Packages

You can fit an object model on a single page for many small and medium-sized problems. However, you will need to organize the presentation of large object models. A person cannot understand a large object model at a glance. Furthermore, it is difficult to get a sense of perspective about the relative importance of portions of a large model.

Caution You must partition a large model to allow comprehension.

A package is a group of elements (classes, associations, generalizations, and lesser packages) with a common theme. A package partitions a model, making it easier to understand and manage. Large applications may require several tiers of packages. Packages form a tree with increasing abstraction toward the root, which is the application, the top-level package.

As Figure 5.4 shows, the notation for a package is a box with the addition of a tab. The purpose of the tab is to suggest the enclosed contents, like a tabbed folder.

Figure 5.4: Notation for a Package

Source: http://www.pearsonhighered.com/assets/hip/us/hip_us_pearsonhighered/samplechapter/0131238299.pdf

Notes

There are various themes for forming packages: dominant classes, dominant relationships, major aspects of functionality, and symmetry.

Example: Many business systems have a Customer package or a Part package; Customer and Part are dominant classes that are important to the business of a corporation and appear in many applications.

Notes An object model of a compiler could be divided into packages for lexical analysis, parsing, semantic analysis, code generation, and optimization.

Did u know? Once some packages have been established, symmetry may suggest additional packages.

On the basis of our experience in creating packages, we can offer the following tips:

- ***Carefully delineate each package's scope:*** The precise boundaries of a package are a matter of judgment. Like other aspects of modeling, defining the scope of a package requires planning and organization. Make sure that class and association names are unique within each package, and use consistent names across packages as much as possible.
- ***Make packages cohesive:*** There should be fewer associations between classes that appear in different packages than between classes that appear in a single package. Classes may appear in multiple packages, helping to bind them, but ordinarily associations and generalizations should appear in a single package.
- ***Define each class in a single package:*** The defining package should show the class name, attributes, and possibly operations. Other packages that refer to a class can use a class icon, a box that contains only the class name. This convention makes it easier to read object diagrams because a class is most prominent in its defining package. It ensures that readers of the object model will not become distracted by possibly inconsistent definitions or be misled by forgetting a prior class definition. This convention also makes it easier to develop packages concurrently.

Task Discuss the purpose of tab in packages.

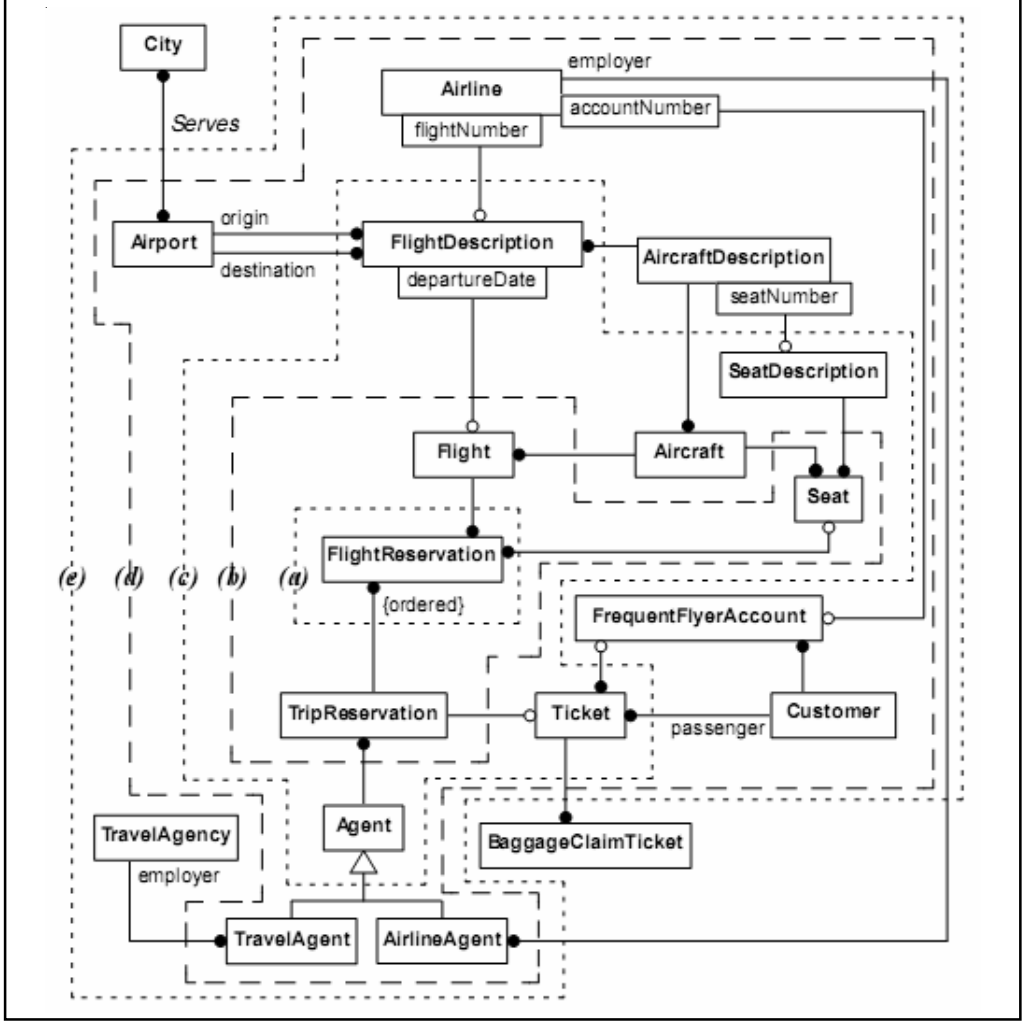
5.4.1 Logical Horizon

You can often use a class with a large logical horizon as the nucleus for a package. The logical horizon of a class is the set of classes reachable by one or more paths terminating in a combined multiplicity of "one" or "zero or one." A path is a sequence of consecutive associations and generalization levels. When computing the logical horizon, you may traverse a generalization hierarchy to obtain further information for a set of objects. You may not, however, traverse to sibling objects, such as by going up and then down the hierarchy. The logical horizons of various classes may, and often do, overlap.

Example: Figure 5.5 shows the computation of the logical horizon for FlightReservation.

Notes

Figure 5.5: Computing the Logical Horizon for FlightReservation



Source: http://www.pearsonhighered.com/assets/hip/us/hip_us_pearsonhighered/samplechapter/0131238299.pdf

- We start with FlightReservation.
- Each FlightReservation has a Flight, Seat and TripReservation.
- A TripReservation implies an Agent and a Ticket. A Flight implies a FlightDescription and an Aircraft. A Seat implies a SeatDescription.
- A Ticket implies a FrequentFlyerAccount and a Customer; an Agent leads to TravelAgent and an AirlineAgent via generalization. The FlightDescription implies an Airport, Airline, and AircraftDescription.
- A TravelAgent has a Travel Agency as an employer. Thus the logical horizon of FlightReservation includes every class in the diagram except City and BaggageClaimTicket.

When computing the logical horizon, you should disregard any qualifiers and treat the associations as if they were unqualified. The purpose of the logical horizon is to compute the objects that can be inferred from some starting object.

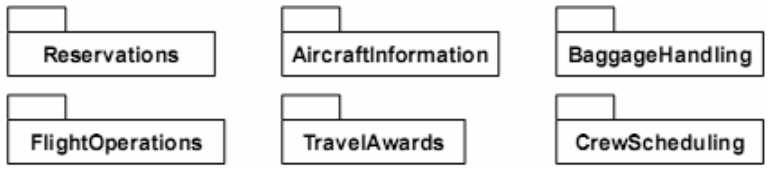
Example of Packages

Notes

Here we will discuss various examples of packages.

Figure 5.6 shows a model for an airline information system with packages organized on a functional basis. The reservations package records customer booking of airline travel. Flight operations deal with the actual logistics of planes arriving and departing. The aircraft information package stores seating layout and manufacturing data. Travel awards tracks bonus free travel for each customer; a person may submit a frequent flyer account number at the time of a reservation, but does not receive credits until after taking the flight. Baggage handling involves managing bags in conjunction with flights and accommodating errant pieces of luggage. Crewscheduling involves scheduling to staff flight needs. The subsequent diagrams elaborate all packages except CrewScheduling.

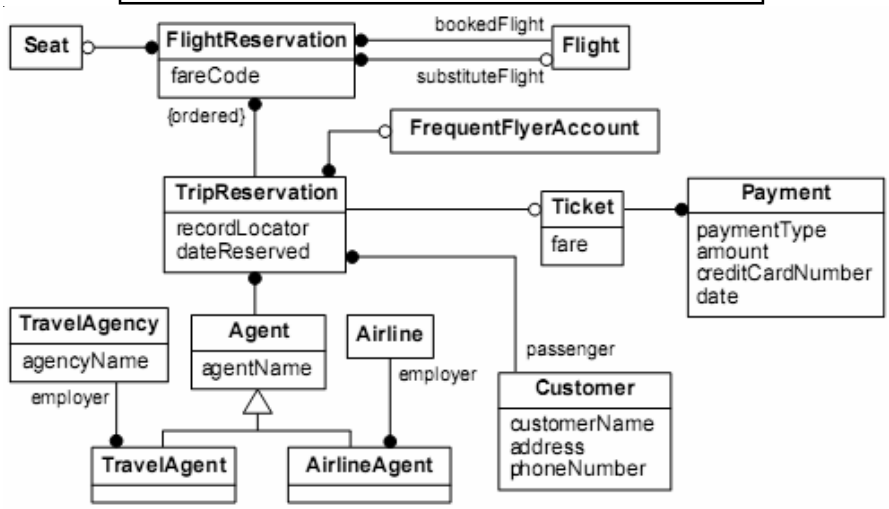
Figure 5.6: A Partial High-level Object Model for an Airline Information System



Source: http://www.pearsonhighered.com/assets/hip/us/hip_us_pearsonhighered/samplechapter/0131238299.pdf

Figure 5.7 describes the reservations package. A trip reservation consists of a sequence of flight reservations, where each flight reservation refers to a specific flight. Sometimes another flight is substituted for a booked flight because of equipment problems, weather delays, or customer preference. The passenger may reserve a seat for each flight. A trip reservation is made on some date; the passenger must purchase a ticket within a certain number of days or the reservation becomes void. The airlines use record locators to find a particular trip reservation quickly and unambiguously. A trip is reserved by an agent, who either works for an airline or a travel agency. The frequent flyer account may be noted for a passenger.

Figure 5.7: An Object Model for the Reservations Package



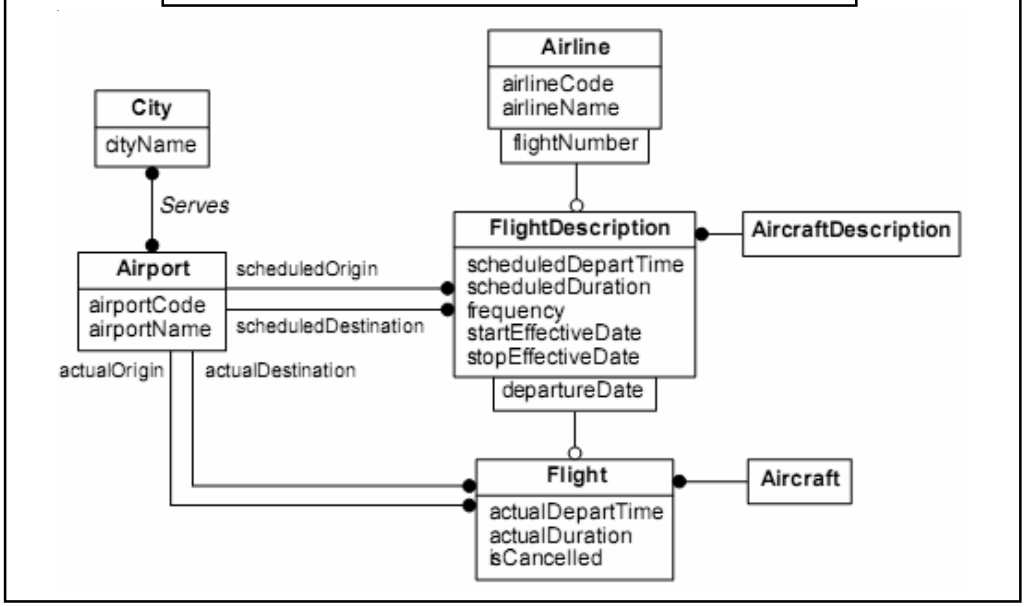
Source: http://www.pearsonhighered.com/assets/hip/us/hip_us_pearsonhighered/samplechapter/0131238299.pdf

Notes

Although the structure of the model does not show it, the owner of the frequent flyer account must be the same as the passenger. We directly associate TripReservation with FrequentFlyerAccount and Customer, because a customer can make a reservation and specify a frequent flyer account before a ticket is even issued. Multiple payments may be made for a trip, such as two credit-card charges. Payment may also be made by cash or check.

Figure 5.8 describes the FlightOperations package. An airport serves many cities, and a city may have multiple airports. Airlines operate flights between airports. A flight description refers to the published description of air travel between two airports. In contrast, a flight refers to the actual travel made by an airplane on a particular date. The frequency indicates the days of the week for which the flight description applies. The start and stop effectivity dates bracket the time period for which the published flight description is in effect. The actual origin, destination, departure time, and duration of a flight can vary because of weather and equipment problems.

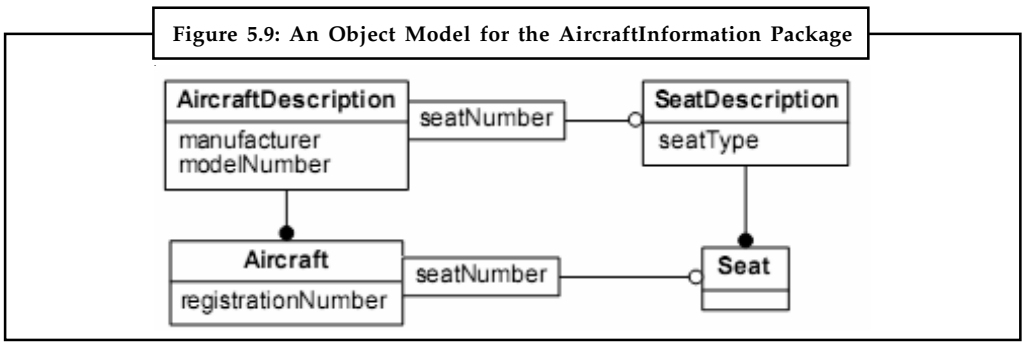
Figure 5.8: An Object Model for the FlightOperations Package



Source: http://www.pearsonhighered.com/assets/hip/us/hip_us_pearsonhighered/samplechapter/0131238299.pdf

Figure 5.9 presents a simple model of the AircraftInformation package. Each aircraft model has a manufacturer, model number, and specific numbering for seats. The seat type may be first class, business, or coach. Each individual aircraft has a registration number and refers to an aircraft model.

Figure 5.9: An Object Model for the AircraftInformation Package

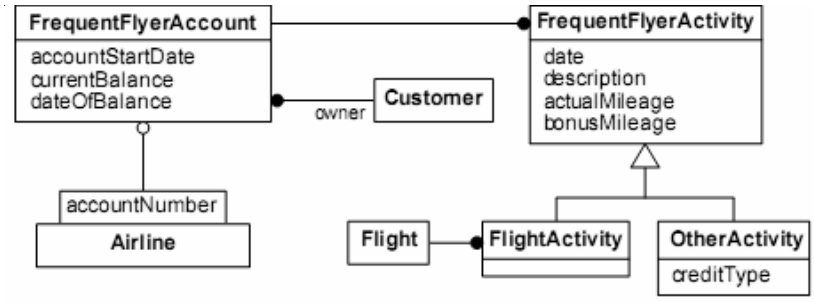


Source: http://www.pearsonhighered.com/assets/hip/us/hip_us_pearsonhighered/samplechapter/0131238299.pdf

Notes

Figure 5.10 describes the TravelAwards package. A customer may have multiple frequent flyer accounts. Airlines identify each account with an account number. An account may receive numerous frequent flyer credits. Some frequent flyer credits pertain to flights; others (indicated by creditType) concern adjustments, redemption, long distance mileage, credit card mileage, hotel stays, car rental, and other kinds of inducements to patronize a business.

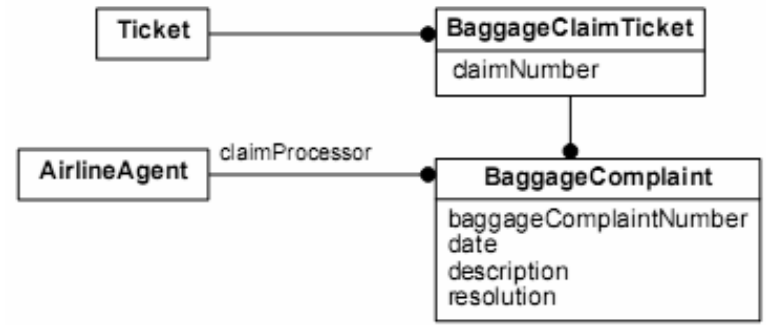
Figure 5.10: An Object Model for the TravelAwards Package



Source: http://www.pearsonhighered.com/assets/hip/us/hip_us_pearsonhighered/samplechapter/0131238299.pdf

Figure 5.11 describes the BaggageHandling package. A customer may check multiple bags for a trip and receives a claim ticket for each bag. Sometimes a bag is lost, damaged, or delayed, in which case the customer completes a baggage complaint form for each problem bag.

Figure 5.11: An Object Model for the BaggageHandling Package



Source: http://www.pearsonhighered.com/assets/hip/us/hip_us_pearsonhighered/samplechapter/0131238299.pdf

Self Assessment

Fill in the blanks:

11. A ..... is a group of elements (classes, associations, generalizations, and lesser packages) with a common theme.
12. The notation for a package is a box with the addition of a .....
13. The ..... of a class is the set of classes reachable by one or more paths terminating in a combined multiplicity of "one" or "zero or one."
14. A ..... is a sequence of consecutive associations and generalization levels.
15. The purpose of the logical horizon is to compute the ..... that can be inferred from some starting object.

Notes

5.5 Summary

- An abstract class is a class that has no direct instances. The descendant classes can also be abstract, but the generalization hierarchy must ultimately terminate in subclasses with direct instances.
- A constraint is a functional relationship between modeling constructs such as classes, attributes, and associations.
- Derived data is data that can be completely determined from other data.
- As a rule, you should not show derived data during analysis unless the data appears in the problem description.
- The notation for derived data is a slash preceding the name of the attribute, class, association, or role.
- A package is a group of elements (classes, associations, generalizations, and lesser packages) with a common theme.
- There are various themes for forming packages: dominant classes, dominant relationships, major aspects of functionality, and symmetry.
- The logical horizon of a class is the set of classes reachable by one or more paths terminating in a combined multiplicity of "one" or "zero or one."

5.6 Keywords

Abstract class: An abstract class is a class that has no direct instances.

Abstract operation: An abstract operation species the signature of an operation while deferring implementation to the subclasses.

Constraint: A constraint is a functional relationship between modeling constructs such as classes, attributes, and associations.

Derived data: Derived data is data that can be completely determined from other data.

Logical Horizon: The logical horizon of a class is the set of classes reachable by one or more paths terminating in a combined multiplicity of "one" or "zero or one."

Package: A package is a group of elements (classes, associations, generalizations, and lesser packages) with a common theme.

Path: A path is a sequence of consecutive associations and generalization levels.

Signature: The signature of an operation species the argument types, the result type, exception conditions, and the semantics of the operation.

5.7 Review Questions

1. Discuss the concept of abstract class with example.
2. What does abstract operation specify? Discuss.
3. What are constraints? Illustrate with example.
4. During design you can add derived data to improve efficiency and ease implementation. Comment.
5. How does a package partitions a model? Discuss.

Notes

6. Analyze various themes used for forming packages.
7. Discuss some tips that should be kept in mind when creating packages.
8. What is the logical horizon of a class? Explain.
9. Compare and contrast the notation used for abstract operations and derived data respectively.
10. Elucidate the computation of the logical horizon with example.

Answers: Self Assessment

1.	abstract class	2.	subclasses
3.	signature	4.	Constraint
5.	data	6.	Text
7.	comments	8.	Derived data
9.	generalization	10.	invalid
11.	package	12.	Tab
13.	logical horizon	14.	Path
15.	objects		

5.8 Further Readings

Books

Booch, Grady, (1994), *Object-oriented Analysis & Design,* Addison Wesley

Laganiere, Robert, (2004), *Object-oriented Software Engineering,* TMH

Rumbaugh, J., (2007), *Object-oriented Modelling and Design with UML,* Pearson Education

Satzinger, (2007), *Object-oriented Analysis & Design with the Unified Process,* Thomson

Online links

http://202.131.97.51/FACULTY/Maitri%20Patel/ooad/solutions%20of%20excercise%20of%20ooad/oo4.pdf

http://comp.mq.edu.au/books/maciaszek/student_resources/Acro_b&w_3/Ch5_b&w_3.pdf

http://www.is.inf.uni-due.de/courses/mod_ws06/skript/uml-ocl.pdf

http://www.scribd.com/doc/27696643/object-oriented-modeling-and-design

Notes

Unit 6: State Modelling

CONTENTS

Objectives

Introduction

6.1 State Machine

6.1.1 Events

6.1.2 States

6.1.3 Transition and Conditions

6.1.4 Action

6.1.5 Activity

6.2 State Diagrams

6.2.1 When to use State Diagrams

6.2.2 How to draw State Diagrams

6.2.3 State Diagram Behaviour

6.3 Summary

6.4 Keywords

6.5 Review Questions

6.6 Further Readings

Objectives

After studying this unit, you will be able to:

- Explain the concept of events and states
- Discuss the concept of transitions and conditions
- Describe State Diagrams
- Explain State Diagram Behaviour

Introduction

The complete OOM revolves around the objects identified in the system. When observed closely, every object exhibits some characteristics and behavior. The objects recognize and respond to certain events. For example, considering a Window on the screen as an object, the size of the window gets changed when resize button of the window is clicked. Here the clicking of the button is an event to which the window responds by changing its state from the old size to the new size. While developing systems based on this approach, the analyst makes use of certain models to analyze and depict these objects. The dynamic model represents a state/transition view on the model. Main concepts are states, transitions between states, and events to trigger transitions. Actions can be modeled as occurring within states. Generalization and aggregation (concurrency) are predefined relationships. The outcomes of a dynamic model are scenarios, event-trace diagrams and state diagrams.

Notes

6.1 State Machine

A state machine is a behavior which specifies the sequence of states an object visits during its lifetime in response to events, together with its responses to those events. Now, we describe the various concepts related to state machine in the following sub-sections.

6.1.1 Events

An event is the specification of a significant occurrence. For a state machine, an event is the occurrence of a stimulus that can trigger a state transition. In other words, we can say an event is something that happens at a point in time. An event does not have duration. An individual stimulus from one object to another is an event.

Example: Press a button on mouse, airplane departs from an airport are examples of events.

6.1.2 States

A state is a condition during the life of an object during which it satisfies some condition, performs some activity, or waits for some external event. A state corresponds to the interval between two events received by an object and describes the "value" of the object for that time period. A state is an abstraction of an object's attribute values and links, where sets of values are grouped together into a state according to properties that affect the general behavior of the object. For instance, stack is empty or stack is full are different states of the object stack. As state corresponds to interval between two events received by an object; therefore, it has duration.

A substate is a state that is nested in another state. A state that has substates is called a composite state. A state that has no substates is called a simple state. Substates may be nested to any level.

6.1.3 Transition and Conditions

A transition is a relationship between two states indicating that an object in the first state will, when a specified set of events and conditions are satisfied, perform certain actions and enter the second state. Transition can be self-transition. It is a transition whose source and target states are the same. If a transition is to a composite state, the nested state machine must have an initial state. If a transition is to a substate, the substate is entered after any entry action for the enclosing composite state is executed followed by any entry action for the substate. If a transition is from a substate within the composite state, any exit action for the substrate is executed followed by any exit action for the enclosing composite state. A transition from the composite state may occur from any of the substates and takes precedence over any of the transitions for the current substate.

A condition is a Boolean function of object values, such as "the temperature is below freezing." A condition is valid over an interval of time.

Example: "The temperature is below freezing from November 15, 1921 until March 3, 1922".

6.1.4 Action

An action is an executable, atomic (with reference to the state machine) computation. Actions may include operations, the creation or destruction of other objects, or the sending of signals to

Notes

other objects (events). An action is an instantaneous operation. An action represents an operation whose duration is insignificant compared to the resolution of the state diagram. For instance, disconnect phone line might be an action in response to an on-hook event for the phone line. An action is associated with an event.

6.1.5 Activity

Activity is an operation that takes time to complete. An activity is associated with a state. Activity includes continuous operations such as displaying a picture on a television screen as well as sequential operations that terminate by them after an interval of time such as closing a valve or performing a computation. A state may control a continuous activity such as ringing a telephone bell that persists until an event terminates it causing the transition of the state. Activity starts on entry to the state and stops on exit.

Did u know? A state may control a sequential activity such as a robot moving a part that progresses until it completes or until it is interrupted by an event that terminates prematurely.

Task Make distinction between events and states.

Self Assessment

Fill in the blanks:

1. An individual stimulus from one object to another is an .....
2. A ..... is an abstraction of an object's attribute values and links.
3. State that has substates is called a ..... state.
4. A state corresponds to the interval between two events received by an object and describes the "....." of the object for that time period.
5. A ..... is a Boolean function of object values.
6. ..... is an operation that takes time to complete.

6.2 State Diagrams

State diagrams are used to describe the behavior of a system. State diagrams describe all of the possible states of an object as events occur. Each diagram usually represents objects of a single class and tracks the different states of its objects through the system.

It relates events and states. A change of state caused by an event is called a transition. Transition is drawn as an arrow from the receiving state to the target state. A state diagram is graph whose nodes are states and whose directed arcs are transitions labeled by event names. State diagram specifies the state sequence caused by an event sequence.

6.2.1 When to use State Diagrams

Use state diagrams to demonstrate the behavior of an object through many use cases of the system. Only use state diagrams for classes where it is necessary to understand the behavior of

Notes

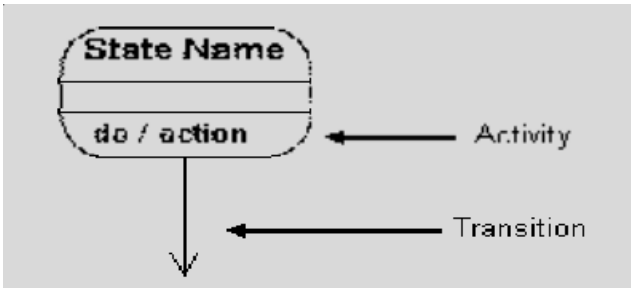
the object through the entire system. Not all classes will require a state diagram and state diagrams are not useful for describing the collaboration of all objects in a use case.

Notes State diagrams are combined with other diagrams such as interaction diagrams and activity diagrams.

6.2.2 How to draw State Diagrams

State diagrams have very few elements. The basic elements are rounded boxes representing the state of the object and arrows indicting the transition to the next state. The activity section of the state symbol depicts what activities the object will be doing while it is in that state as shown Figure 6.1 below.

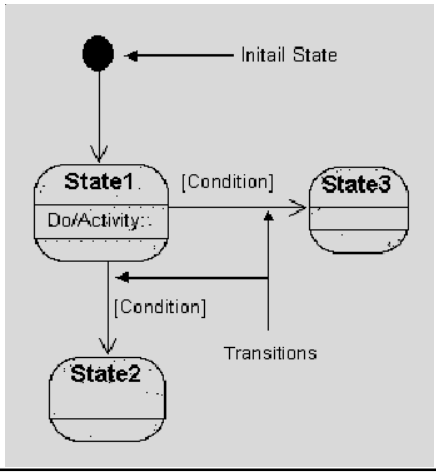
Figure 6.1: Activity Section



Source: http://www.ddegjust.ac.in/studymaterial/mca-5/mca-503.pdf

Initial and Final States: All state diagrams being with an initial state of the object as shown in Figure 6.2. This is the state of the object when it is created. After the initial state the object begins changing states. Conditions based on the activities can determine what the next state the object transitions to.

Figure 6.2: Initial State



Source: http://www.ddegjust.ac.in/studymaterial/mca-5/mca-503.pdf

The initial state is denoted by a filled black circle and may be labeled with a name. The final state is denoted by a circle with a dot inside and may also be labeled with a name as shown in Figure 6.3.

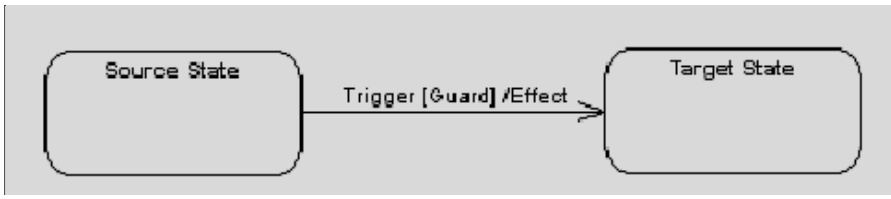
Notes

Figure 6.3: Final State

Source: http://www.ddegjust.ac.in/studymaterial/mca-5/mca-503.pdf

Transitions: Transitions from one state to the next are denoted by lines with arrowheads. A transition may have a trigger, a guard and an effect, as shown in Figure 6.4 below.

Figure 6.4: Transitions



Source: http://www.ddegjust.ac.in/studymaterial/mca-5/mca-503.pdf

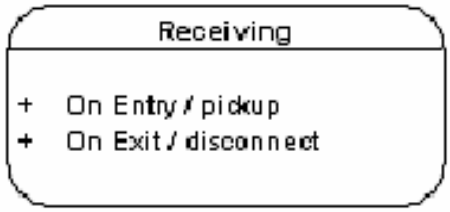
"Trigger" is the cause of the transition, which could be a signal, an event, a change in some condition, or the passage of time.

Notes "Guard" is a condition which must be true in order for the trigger to cause the transition. "Effect" is an action which will be invoked directly on the object that owns the state machine as a result of the transition.

State Actions: In the transition example above, an effect was associated with the transition. If the target state had many transitions arriving at it, and each transition had the same effect associated with it, it would be better to associate the effect with the target state rather than the transitions. This can be done by defining an entry action for the state.

Example: The diagram in Figure 6.5 below shows a state with an entry action and an exit action

Figure 6.5: State with an Entry Action and an Exit Action



Source: http://www.ddegjust.ac.in/studymaterial/mca-5/mca-503.pdf

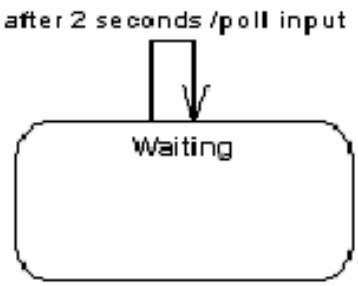
It is also possible to define actions that occur on events, or actions that always occur.

Notes

Caution It is possible to define any number of actions of each type.

Self-transitions: A state can have a transition that returns to itself as shown in the Figure 6.6. This is the most useful when an effect is associated with the transition.

Figure 6.6: Self-transition

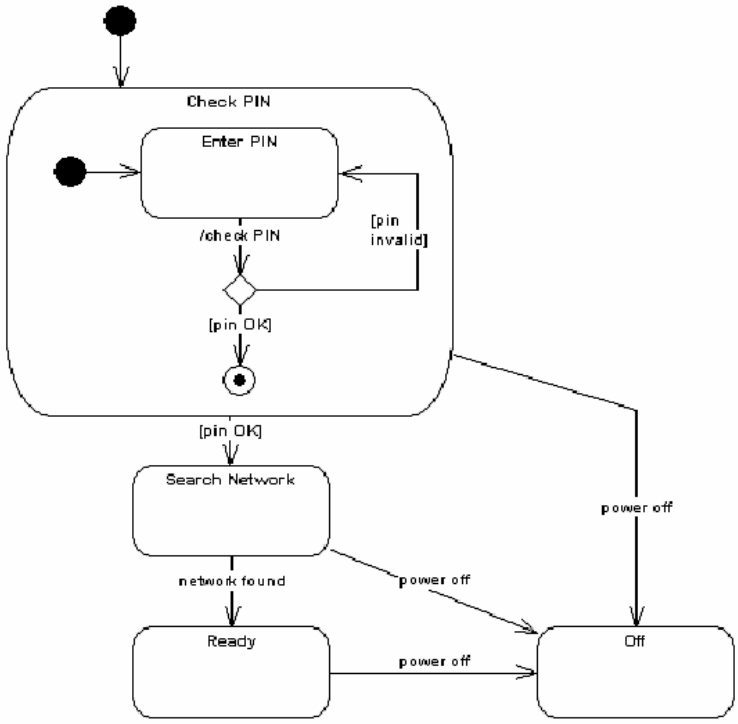


Source: http://www.ddegjust.ac.in/studymaterial/mca-5/mca-503.pdf

Compound States: Substates may be nested to any level.

Example: In figure 6.7, we have shown that a state machine diagram may include sub-machine diagrams.

Figure 6.7: Compound States



Source: http://www.ddegjust.ac.in/studymaterial/mca-5/mca-503.pdf

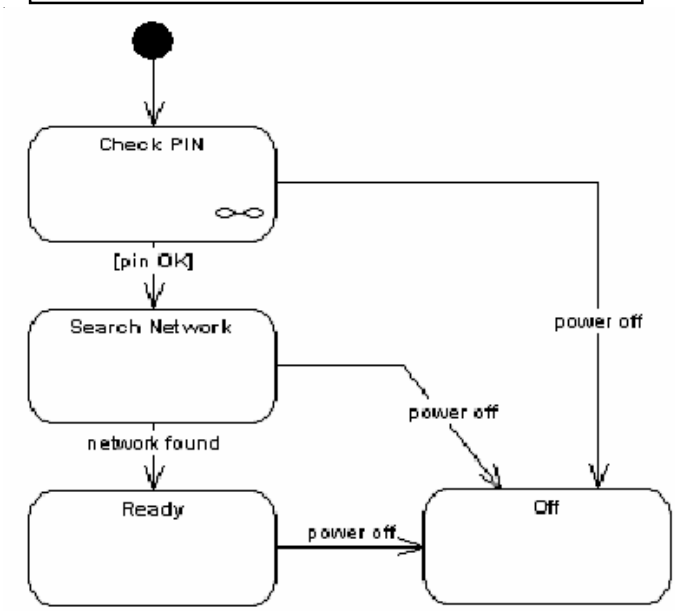
There is also an alternative way of showing the same information.

Notes

Caution The notation in the above version indicates that the details of the Check PIN sub-machine are shown in a separate diagram.

Example: The alternative way to show the same information is shown in Figure 6.8.

Figure 6.8: Alternate Way of showing Compound States

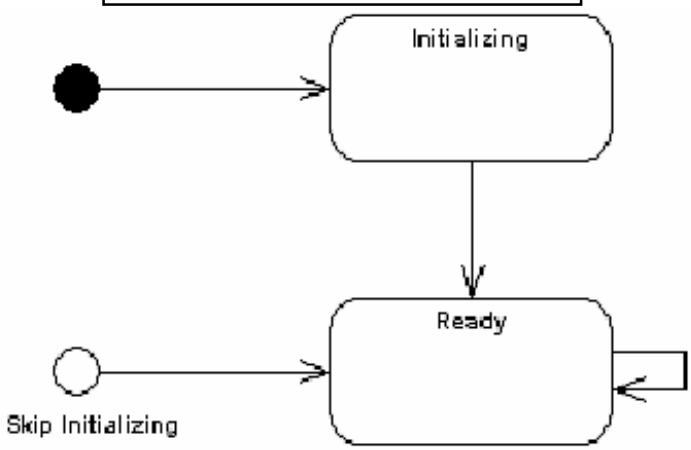


Source: http://www.ddegjust.ac.in/studymaterial/mca-5/mca-503.pdf

Entry Point: Sometimes you won't want to enter a submachine at the normal initial state.

Example: In the submachine shown in Figure 6.9, it would be normal to begin in the "Initializing" state, but if for some reason it wasn't necessary to perform the initialization, it would be possible to begin in the "Ready" state by transitioning to the named entry point.

Figure 6.9: Entry Point Example



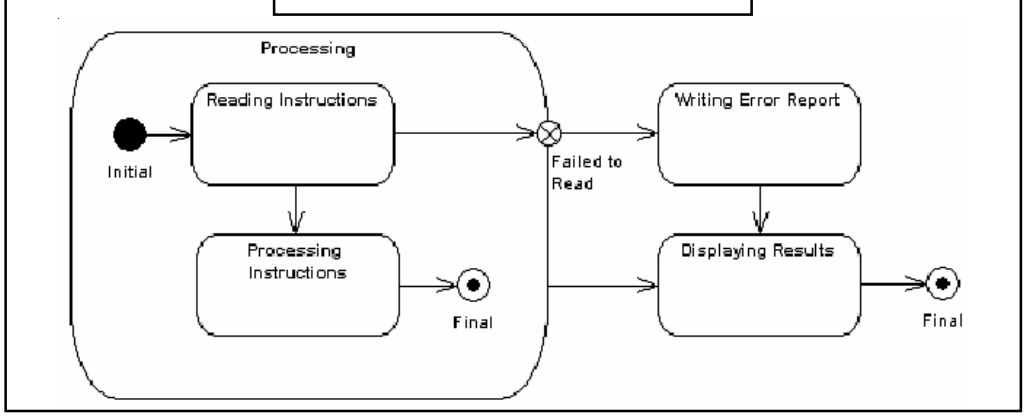
Source: http://www.ddegjust.ac.in/studymaterial/mca-5/mca-503.pdf

Notes

Exit Point: In a similar manner to entry points, it is possible to have named alternative exit points.

Example: In Figure 6.10 gives an example where the state executed after the main processing state depends on which route is used to transition out of the state.

Figure 6.10: Exit Point Example



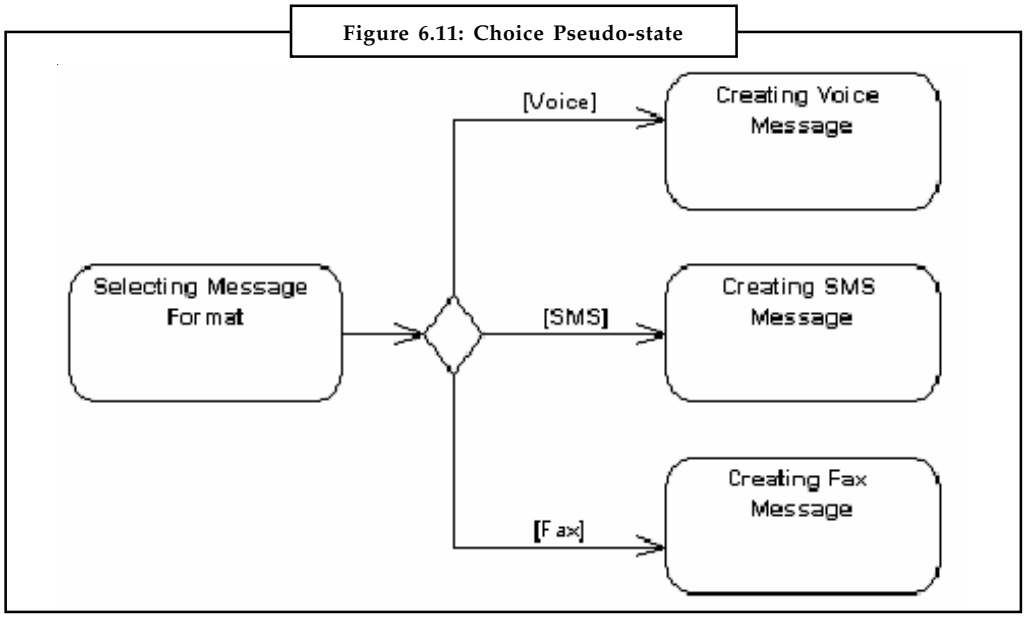
Source: http://www.ddegjust.ac.in/studymaterial/mca-5/mca-503.pdf

Task Compare and contrast entry point and exit point.

Choice Pseudo-State: A choice pseudo-state is shown as a diamond with one transition arrives and two or more transitions leaving.

Example: In Figure 6.11 shows that whichever state is arrived at, after the choice pseudo-state, is dependent on the message format selected during execution of the previous state.

Figure 6.11: Choice Pseudo-state



Source: http://www.ddegjust.ac.in/studymaterial/mca-5/mca-503.pdf

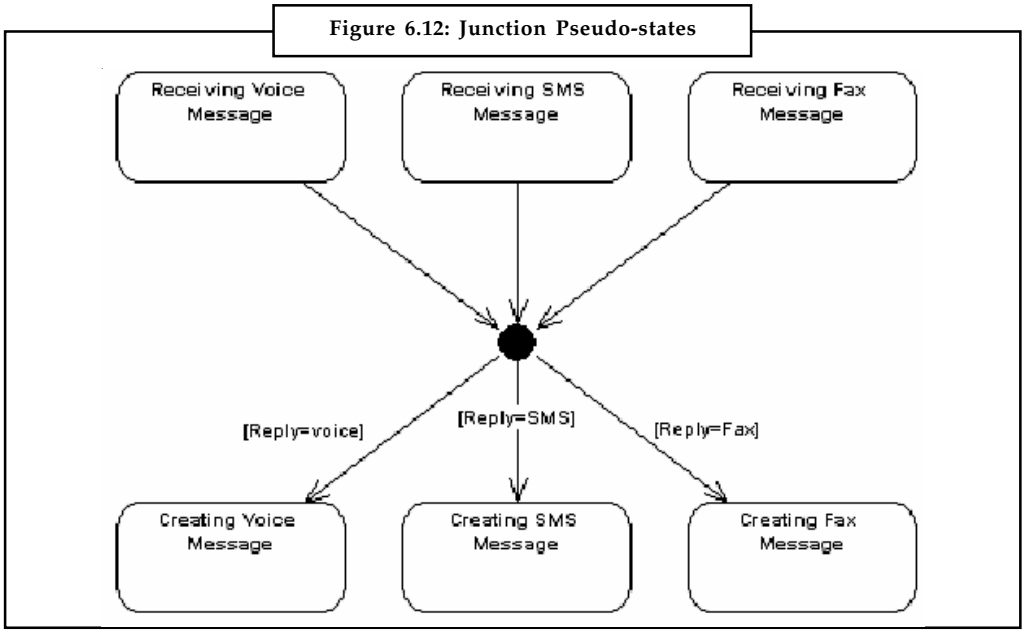
Notes

Junction Pseudo-State: Junction pseudo-states are used to chain together multiple transitions. A single junction can have one or more incoming, and one or more outgoing, transitions; a guard can be applied to each transition. Junctions are semantic-free.

Did u know? A junction, which splits an incoming transition into multiple outgoing transitions realizes a static conditional branch, as opposed to a choice pseudo-state which realizes a dynamic conditional branch.

Example: Junction pseudo-states is shown in Figure 6.12.

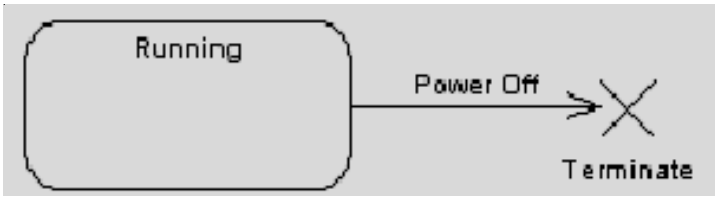
Figure 6.12: Junction Pseudo-states



Source: http://www.ddegjust.ac.in/studymaterial/mca-5/mca-503.pdf

Terminate Pseudo-State: Entering a terminate pseudo-state indicates that the lifeline of the state machine has ended. A terminate pseudo-state is denoted as a cross as shown in Figure 6.13.

Figure 6.13: Terminate Pseudo-state



Source: http://www.ddegjust.ac.in/studymaterial/mca-5/mca-503.pdf

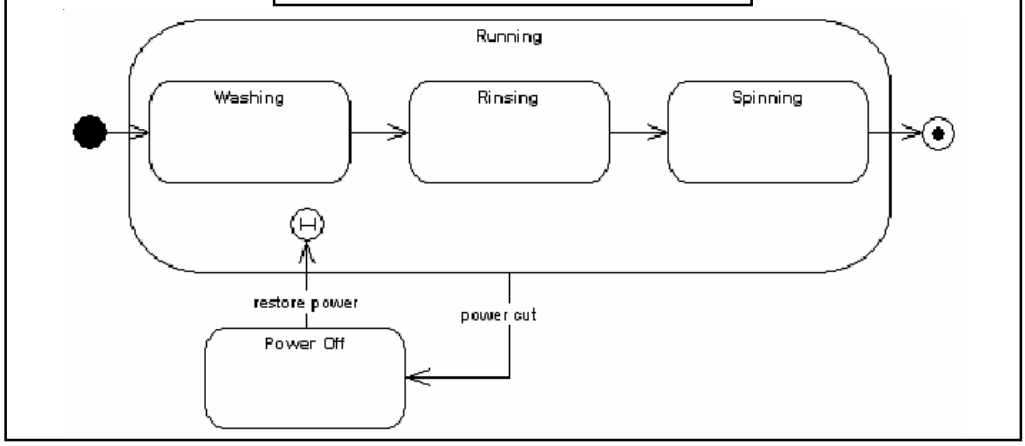
History States: A history state is used to remember the previous state of a state machine when it was interrupted.

Example: The diagram shown in Figure 6.14 illustrates the use of history states. The example is a state machine belonging to a washing machine. In this state machine, when a washing machine is running, it will progress from "Washing" through "Rinsing" to "Spinning".

Notes

If there is a power cut, the washing machine will stop running and will go to the "Power Off" state. Then when the power is restored, the Running state is entered at the "History State" symbol meaning that it should resume where it last left-off.

Figure 6.14: Use of History States

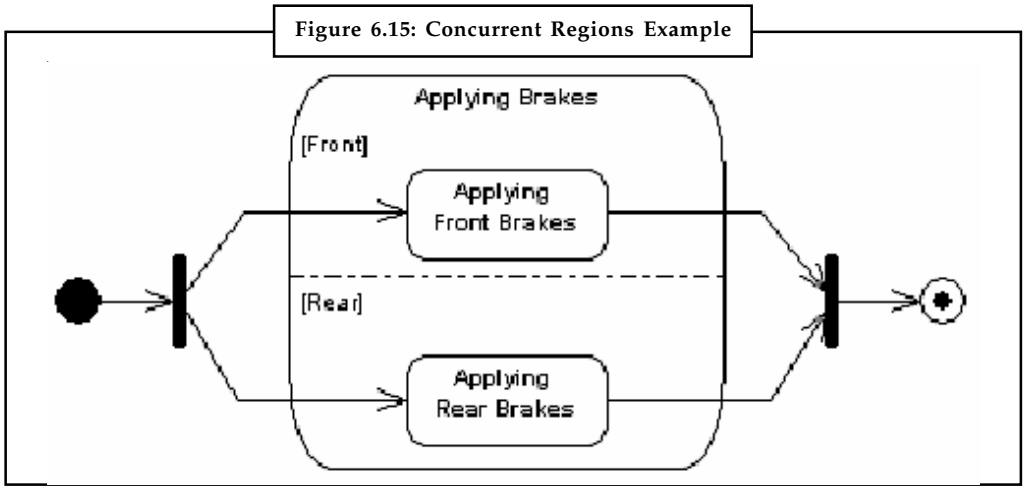


Source: http://www.ddegjust.ac.in/studymaterial/mca-5/mca-503.pdf

Concurrent Regions: A state may be divided into regions containing substates that exist and execute concurrently.

Example: Figure 6.15 shows that within the state "Applying Brakes", the front and rear brakes will be operating simultaneously and independently. Notice the use of fork and join pseudo-states, rather than choice and merge pseudo-states. These symbols are used to synchronize the concurrent threads.

Figure 6.15: Concurrent Regions Example



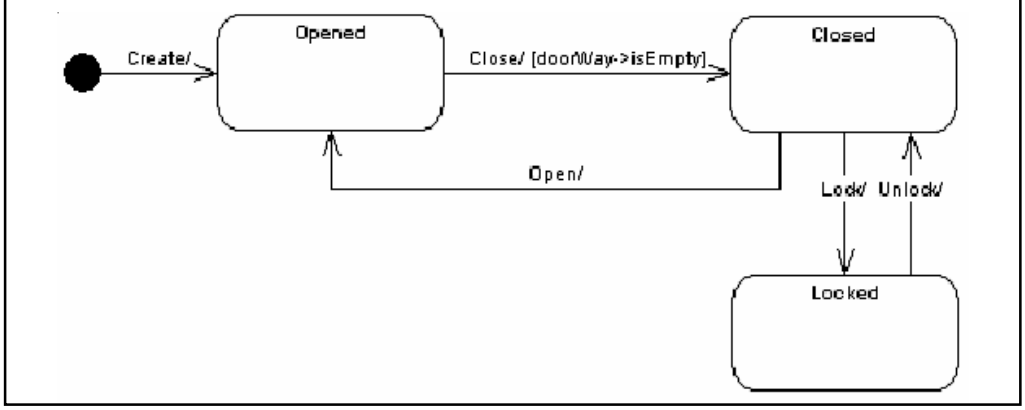
Source: http://www.ddegjust.ac.in/studymaterial/mca-5/mca-503.pdf

A state machine diagram models the behavior of a single object, specifying the sequence of events that an object goes through during its lifetime in response to events.

Example: The following state machine in Figure 6.16 shows the states that a door goes through during its lifetime.

Notes

Figure 6.16: State Machine with three States: "Opened", "Closed" or "Locked"

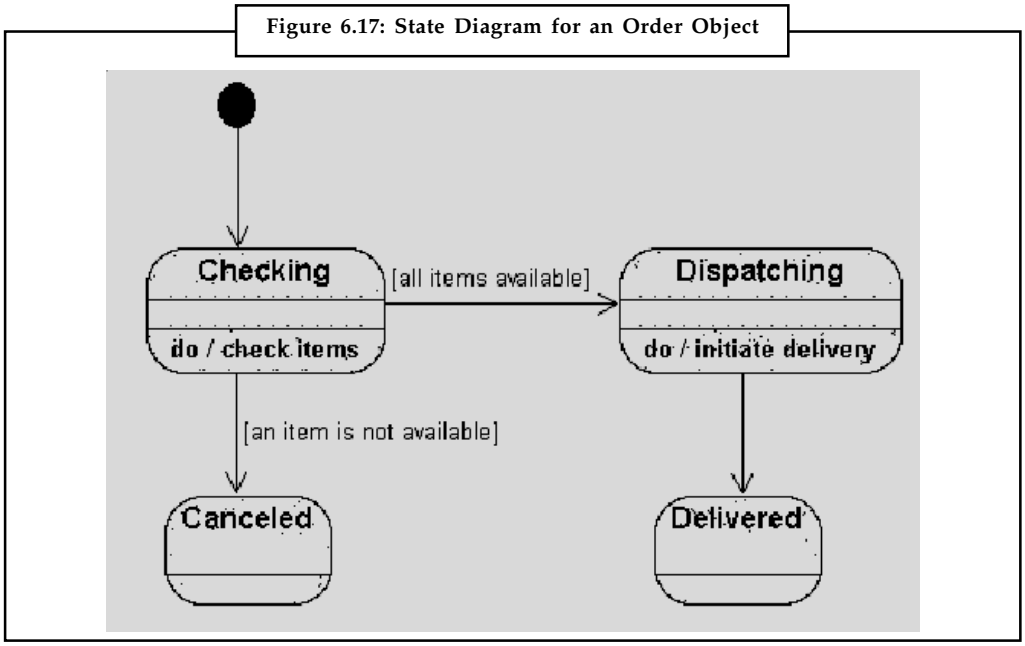


Source: http://www.ddegjust.ac.in/studymaterial/mca-5/mca-503.pdf

The door can be in one of three states: "Opened", "Closed" or "Locked". It can respond to the events Open, Close, Lock and Unlock. Notice that not all events are valid in all states; for example, if a door is opened, you cannot lock it until you close it. Also notice that a state transition can have a guard condition attached: if the door is Opened, it can only respond to the Close event if the condition door Way->isEmpty is fulfilled.

Example: Let us take another example of the state diagram for an Order object as shown in Figure 6.17. When the object enters the Checking state it performs the activity "check items." After the activity is completed the object transitions to the next state based on the conditions [all items available] or [an item is not available]. If an item is not available the order is canceled. If all items are available then the order is dispatched. When the object transitions to the Dispatching state the activity "initiate delivery" is performed. After this activity is complete the object transitions again to the Delivered state.

Figure 6.17: State Diagram for an Order Object



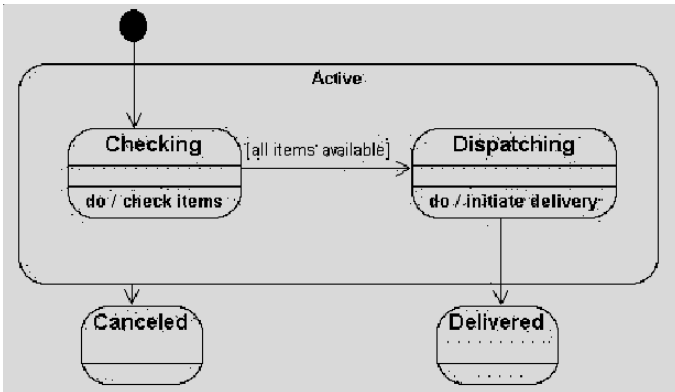
Source: http://www.ddegjust.ac.in/studymaterial/mca-5/mca-503.pdf

Notes

State diagrams can also show a superstate for the object. A superstate is used when many transitions lead to a certain state. Instead of showing all of the transitions from each state to the redundant state a superstate can be used to show that all of the states inside of the superstate can transition to the redundant state. This helps make the state diagram easier to read.

Example: Figure 6.18 shows a superstate. Both the Checking and Dispatching states can transition into the Canceled state, so a transition is shown from a superstate named Active to the state Cancel. By contrast, the state Dispatching can only transition to the Delivered state, so we show an arrow only from the Dispatching state to the Delivered state.

Figure 6.18: Superstate

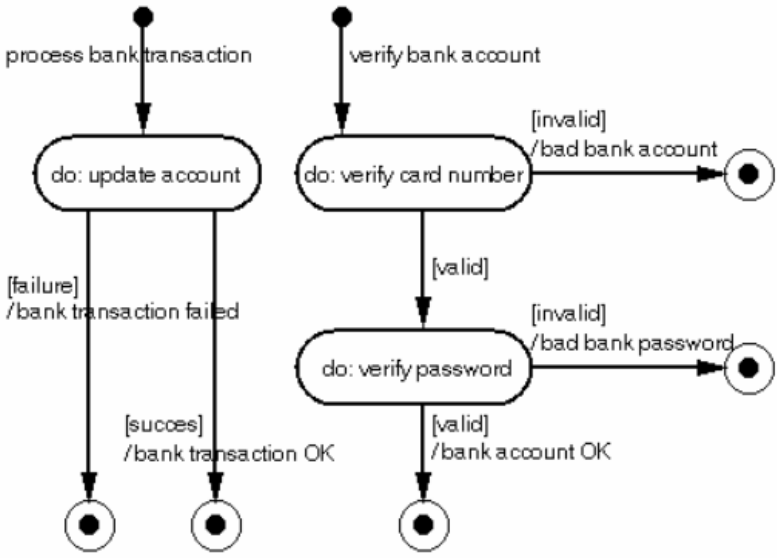


Source: http://www.ddegjust.ac.in/studymaterial/mca-5/mca-503.pdf

Another illustrative example of state diagram is shown below.

Example: Figure 6.19 shows the bank transaction and verification of bank account. Activities are shown in the state such as do: update account, do: verify card number and do: verify password. Diagram also shows initial and final states.

Figure 6.19: State Diagram of Bank Transaction and Verification of Bank Account



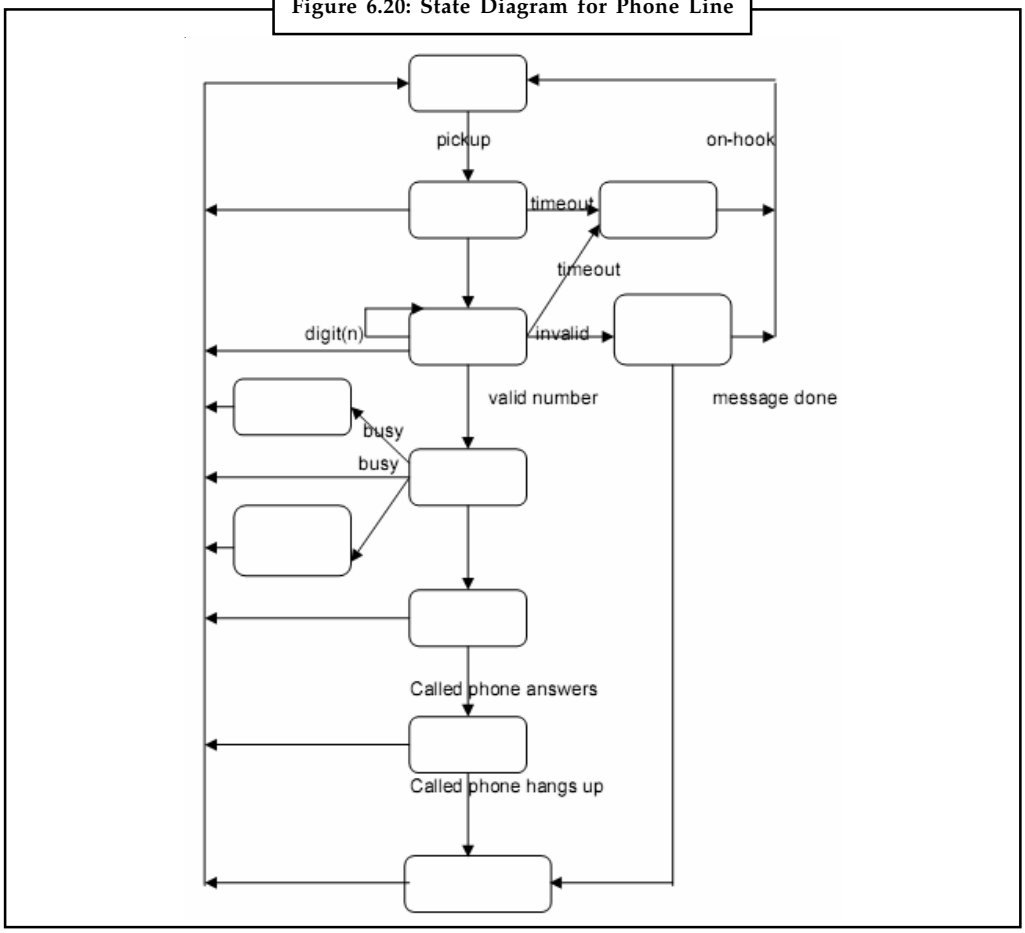
Source: http://www.ddegjust.ac.in/studymaterial/mca-5/mca-503.pdf

Notes

Let us discuss one more example of state diagram.

Example: Now, we consider one more example of state diagram for phone line is shown Figure 6.20. A phone can have many states such as idle, dial tone, dialing, connecting ringing, etc. various states and events are shown in figure below.

Figure 6.20: State Diagram for Phone Line



Source: http://www.ddegjust.ac.in/studymaterial/mca-5/mca-503.pdf

6.2.3 State Diagram Behaviour

As we have discussed above that state diagrams are used to demonstrate the behavior of an object through many use cases of the system. Now let us show an example of state diagram behavior.

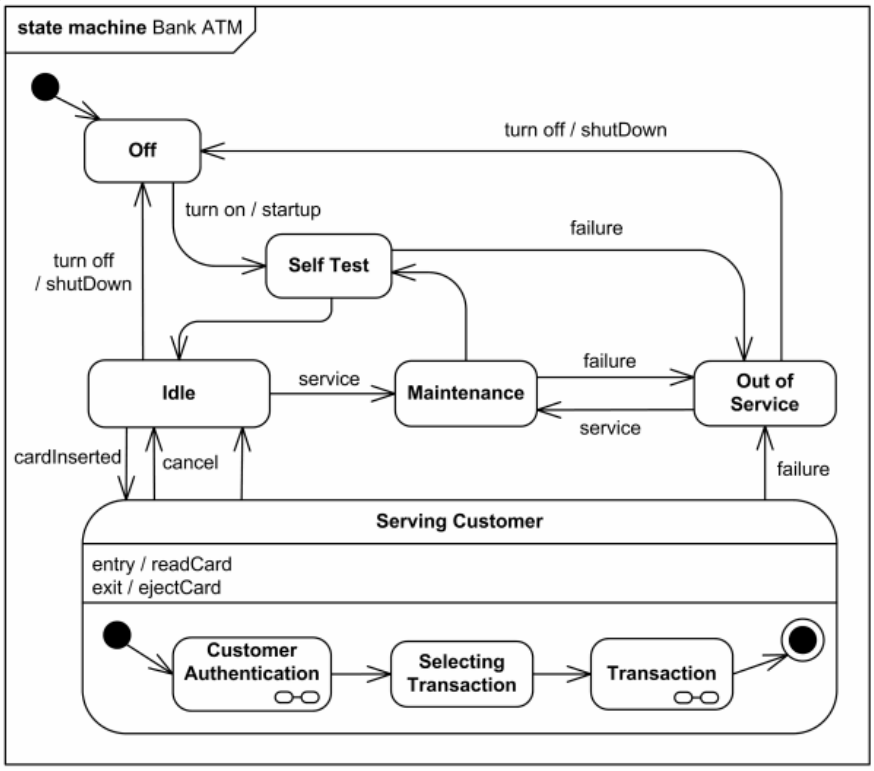
Example: This is an example of UML behavioral state machine diagram showing Bank Automated Teller Machine (ATM) top level state machine. ATM is initially turned off. After the power is turned on, ATM performs start-up action and enters **Self Test** state. If the test fails, ATM goes into **Out of Service** state; otherwise there is **triggerless transition** to the **Idle** state. In this state ATM waits for customer interaction.

The ATM state changes from **Idle** to **Serving Customer** when the customer inserts banking or credit card in the ATM's card reader. On entering the **Serving Customer** state, the entry action

Notes

readCard is performed. Note, that transition from **Serving Customer** state back to the **Idle** state could be triggered by cancel event as the customer could cancel transaction at any time.

Figure 6.21: Behavioral State Machine Example - Bank ATM



Source: http://www.uml-diagrams.org/examples/bank-atm-example.html

Serving Customer state is a composite state with sequential substates **Customer Authentication**, **Selecting Transaction** and **Transaction**. **Customer Authentication** and **Transaction** are composite states by themselves which is shown with hidden decomposition indicator icon. **Serving Customer** state has **triggerless transition** back to the **Idle** state after transaction is finished. The state also has exit action **ejectCard** which releases customer's card on leaving the state, no matter what caused the transition out of the state.

Self Assessment

Fill in the blanks:

7. ..... describe all of the possible states of an object as events occur.
8. A .....pseudo-state is shown with one transition arrives and two or more transitions leaving.
9. ..... pseudo-states are used to chain together multiple transitions.
10. Entering a ..... pseudo-state indicates that the lifeline of the state machine has ended.
11. A ..... state is used to remember the previous state of a state machine when it was interrupted.
12. ..... are used to demonstrate the behavior of an object through many use cases of the system.

Notes

13. A ..... is used when many transitions lead to a certain state.
14. State may be divided into regions containing ..... that exist and execute concurrently.
15. ..... from one state to the next are denoted by lines with arrowheads.

Case Study

State Diagram for Telephone Line

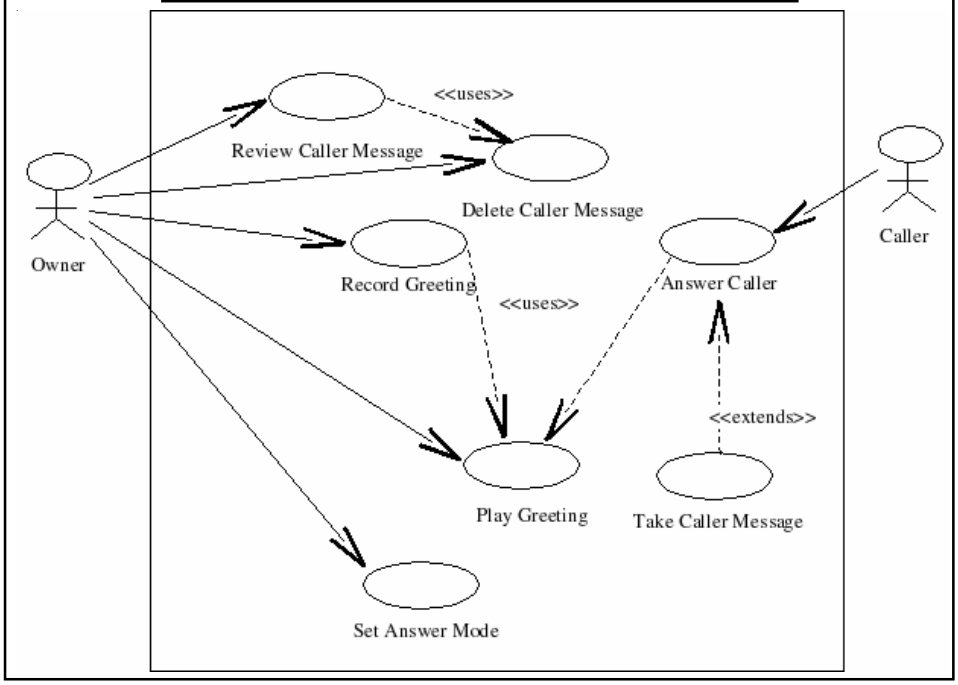
Consider the class for telephone line with following activities and states

- As a start of a call, the telephone line is idle. When the phone receiver is picked from hook, it gives a dial tone and can accept the dialing of digits.
- If after getting dial tone, if the user doesn't dial number within time interval then time out occurs and phone line gets idle.
- After dialing a number, if the number is invalid then some recorded message is played.

  Upon entry of a valid number, the phone system tries to connect a call & routes it to proper destination.

  If the called person answers the phone, the conversation can occur. When called person hangs up, the phone disconnects and goes to idle state.
- Draw the state transition diagram for above description of telephone line.

Figure 1: State Transition Diagram of Telephone Line



Question

Explain the use case relationships used in the state transition diagram of telephone line.

Source: http://www.programsformca.com/2012/03/state-diagram-for-telephone-line.html

Notes

6.3 Summary

- An event is the specification of a significant occurrence. For a state machine, an event is the occurrence of a stimulus that can trigger a state transition.
- A state is a condition during the life of an object during which it satisfies some condition, performs some activity, or waits for some external event.
- A transition is a relationship between two states indicating that an object in the first state will, when a specified set of events and conditions are satisfied, perform certain actions and enter the second state.
- A condition is a Boolean function of object values. A condition is valid over an interval of time.
- Activity includes continuous operations such as displaying a picture on a television screen as well as sequential operations that terminate by themselves after an interval of time such as closing a valve or performing a computation.
- State diagrams are used to describe the behavior of a system. State diagrams describe all of the possible states of an object as events occur.
- State diagrams have very few elements. The basic elements are rounded boxes representing the state of the object and arrows indicting the transition to the next state.
- State diagrams are used to demonstrate the behavior of an object through many use cases of the system.

6.4 Keywords

Action: An action is an executable, atomic (with reference to the state machine) computation.

Activity: Activity is an operation that takes time to complete.

Condition: A condition is a Boolean function of object values.

Event: An event is the specification of a significant occurrence.

State diagrams: State diagrams describe all of the possible states of an object as events occur.

State machine: A state machine is a behavior which specifies the sequence of states an object visits during its lifetime in response to events, together with its responses to those events.

State: A state is a condition during the life of an object during which it satisfies some condition, performs some activity, or waits for some external event.

Transition: A transition is a relationship between two states indicating that an object in the first state will, when a specified set of events and conditions are satisfied, perform certain actions and enter the second state.

6.5 Review Questions

1. Illustrate the concept of events and states with example.
2. Make distinction between transition and condition.
3. What are state diagrams? Illustrate the use of state diagrams.
4. Elucidate the concept of initial and final states in the state diagram.
5. What are state actions? Illustrate with example.

Notes

6. "A state machine diagram may include submachine diagrams." Comment.
7. Discuss the concept of entry point and exit point. Explain with figure.
8. Make distinction between choice Pseudo-State and Junction Pseudo-State.
9. Show how a state can be divided into regions containing substates that exist and execute concurrently.
10. Explain the concept of state diagram behaviour with example.

Answers: Self Assessment

1.	Event	2.	State
3.	composite	4.	Value
5.	condition	6.	Activity
7.	State diagrams	8.	choice
9.	Junction	10.	terminate
11.	History	12.	State diagrams
13.	superstate	14.	substates
15.	Transitions		

6.6 Further Readings

Books

Booch, Grady (1994), *Object-oriented Analysis & Design,* Addison Wesley

Laganiere, Robert, (2004), *Object-oriented Software Engineering,* TMH

Rumbaugh, J. (2007), *Object-oriented Modelling and Design with UML,* Pearson Education

Satzinger, (2007), *Object-oriented Analysis & Design with the Unified Process,* Thomson

Online links

http://debian.fmi.uni-sofia.bg/~vls/public/dsmt/uml/Lectures2/ooad10.pdf

http://www.imse.hku.hk/imse1013/pdf/ESA-06%20OOAD.pdf

http://www.slideshare.net/niitstudentcare/09-ooad-uml11

http://www8.cs.umu.se/~c02pon/oosd2/workbook/OOAD/sm.php

Notes

Unit 7: Interaction Modelling

CONTENTS

Objectives

Introduction

7.1 Use Case Models

7.1.1 Use Case Concept

7.1.2 Basic Model Elements

7.1.3 Use Case Diagrams

7.2 Sequence Models

7.2.1 Basic Sequence Diagram Symbols and Notations

7.3 Activity Models

7.3.1 Basic Activity Diagram Symbols and Notations

7.4 Summary

7.5 Keywords

7.6 Review Questions

7.7 Further Readings

Objectives

After studying this unit, you will be able to:

- Describe the concept of use case models
- Discuss the basic use case model elements
- Explain the concept of sequence models
- Analyse the concept of activity models

Introduction

Interaction diagrams are those diagrams that explain the way of getting the job done by collaboration of groups of object. These diagrams take into custody the behaviour of a single use case, presentation the pattern of interaction among objects. Interaction diagrams describe how groups of objects collaborate to get the job done. Interaction diagrams capture the behavior of a single use case, showing the pattern of interaction among objects. The purpose of Interaction diagrams is to model interactions between objects, assist in understanding how a system (a use case) actually works, verify that a use case description can be supported by the existing classes Identify responsibilities/operations and assign them to classes. These allow you to work out how your classes can help the user solve problems. Sequence diagrams make the timing of various operations clear.

Notes

7.1 Use Case Models

A use case model is a model of how different types of users interact with the system to solve a problem. As such, it describes the goals of the users, the interactions between the users and the system, and the required behavior of the system in satisfying these goals.

A use case model consists of a number of model elements. The most important model elements are: use cases, actors and the relationships between them.

A use case defines a goal-oriented set of interactions between external actors and the system under consideration. Actors are parties outside the system that interact with the system. An actor may be a class of users, roles users can play, or other systems. Cockburn (1997) distinguishes between primary and secondary actors. A primary actor is one having a goal requiring the assistance of the system. A secondary actor is one from which the system needs assistance.

A use case is initiated by a user with a particular goal in mind, and completes successfully when that goal is satisfied. It describes the sequence of interactions between actors and the system necessary to deliver the service that satisfies the goal. It also includes possible variants of this sequence,

Example: Alternative sequences that may also satisfy the goal, as well as sequences that may lead to failure to complete the service because of exceptional behavior, error handling, etc.

The system is treated as a "black box", and the interactions with system, including system responses, are as perceived from outside the system.

Thus, use cases capture who (actor) does what (interaction) with the system, for what purpose (goal), without dealing with system internals. A complete set of use cases specifies all the different ways to use the system, and therefore defines all behavior required of the system, bounding the scope of the system.

Notes Generally, use case steps are written in an easy-to-understand structured narrative using the vocabulary of the domain. This is engaging for users who can easily follow and validate the use cases, and the accessibility encourages users to be actively involved in defining the requirements.

A scenario is an instance of a use case, and represents a single path through the use case. Thus, one may construct a scenario for the main flow through the use case, and other scenarios for each possible variation of flow through the use case.

Example: Triggered by options, error conditions, security breaches, etc.

Scenarios may be depicted using sequence diagrams.

Use case is a representation of a user goal to be satisfied. A system can be considered a collection of use cases together represented in a use case model. The use case model is a picture intended to be easily 'surveyable' and changeable by customers and developers alike. A use case model (UCM) has actors, task ovals, associations and a system boundary. UCMs start simple and become more complex over time. Each use case has two parts; a graphical representation and a textural representation.

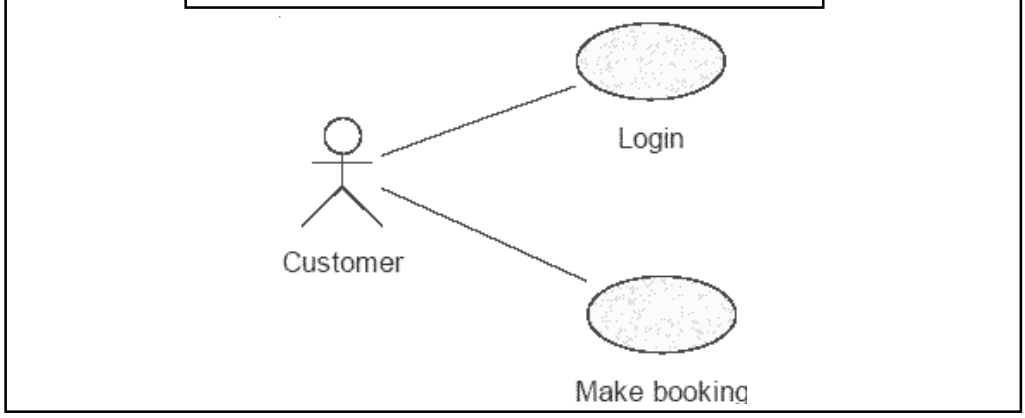
The text part adds detail to the graphical representation.

Notes

It is convenient to consider the graphical component of a use case as a kind of table of contents that directs the reader to the accompanying text.

Example: In Figure 7.1, the most basic form of a use case is represented graphically by a named task oval that represents a user goal. This diagram illustrates a customer's ability to Login and to make booking. As more detail becomes available, it is added to the use case.

Figure 7.1: A Simple use Case Model, Featuring an Actor, Two Task Ovals, with Associations



Caution Over time, the use case becomes primarily a textural construct that describes the system behaviour in semi-formalised natural language.

7.1.1 Use Case Concept

A use case diagram is used to graphically depict a subset of the model to simplify communications. There will typically be several use case diagrams associated with a given model, each showing a subset of the model elements relevant for a particular purpose. The same model element may be shown on several use case diagrams, but each instance must be consistent. If tools are used to maintain the use case model, this consistency constraint is automated so that any changes to the model element will be automatically reflected on every use case diagram that shows that element.

The use case model may contain packages that are used to structure the model to simplify analysis, communications, navigation, development, maintenance and planning.

Much of the use case model is in fact textual, with the text captured in the use case specifications that are associated with each use case model element. These specifications describe the flow of events of the use case.

The use case model serves as a unifying thread throughout system development. It is used as the primary specification of the functional requirements for the system, as the basis for analysis and design, as an input to iteration planning, as the basis of defining test cases and as the basis for user documentation.

7.1.2 Basic Model Elements

The use case model contains, as a minimum, the following basic model elements:

- ***Actor:*** A model element representing each actor. Properties include the actors name and brief description.

Notes

- ***Use Case:*** A model element representing each use case. Properties include the use case name and use case specification.
- ***Associations:*** Associations are used to describe the relationships between actors and the use cases they participate in. This relationship is commonly known as a "communicates-association".
- ***Advanced model elements:*** The use-case model may also contain the following advanced model elements.
- ***Subject:*** A model element that represents the boundary of the system of interest.
- ***Use Case Package:*** A model element used to structure the use case model to simplify analysis, communications, navigation, and planning. If there are many use cases or actors, you can use use case packages to further structure the use case model in much the same manner you use folders or directories to structure the information on your hard-disk.

  You can partition a use case model into use case packages for several reasons, including:

  - To reflect the order, configuration, or delivery units in the finished system thus supporting iteration planning.
  - To support parallel development by dividing the problem into bite-sized pieces.
  - To simplify communication with different stakeholders by creating packages for containing use cases and actors relevant to a particular stakeholder.
- ***Generalisations:*** A relationship between actors to support reuse of common properties.
- ***Dependencies:*** A number of dependency types between use cases are defined in UML. In particular, <<extend>> and <<include>>.

  - <<extend>> is used to include optional behavior from an extending use case in an extended use case.
  - <<include>> is used to include common behavior from an included use case into a base use case in order to support reuse of common behavior.

  The latter is the most widely used dependency and is useful for:

  - Factoring out behavior from the base use case that is not necessary for the understanding of the primary purpose of the use case to simplify communications.
  - Factoring out behavior that is in common for two or more use cases to maximise reuse, simplify maintenance and ensure consistency.

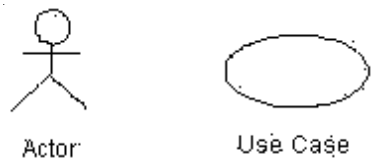
Task Make distinction between <<extend>> and <<include>>.

7.1.3 Use Case Diagrams

A use case is a set of scenarios that describing an interaction between a user and a system. A use case diagram displays the relationship among actors and use cases. The two main components of a use case diagram are use cases and actors.

Notes

Figure 7.2: Components of Use Case Diagram



An actor represents a user or another system that will interact with the system you are modeling.

Did u know? A use case is an external view of the system that represents some action the user might perform in order to complete a task.

Now we will discuss when to Use Cases Diagrams.

Use cases are used in almost every project. These are helpful in exposing requirements and planning the project. During the initial stage of a project most use cases should be defined, but as the project continues more might become visible.

Now let us see how to draw Use Cases Diagrams.

Use cases are a relatively easy UML diagram to draw, but this is a very simplified example. This example is only meant as an introduction to the UML and use cases.

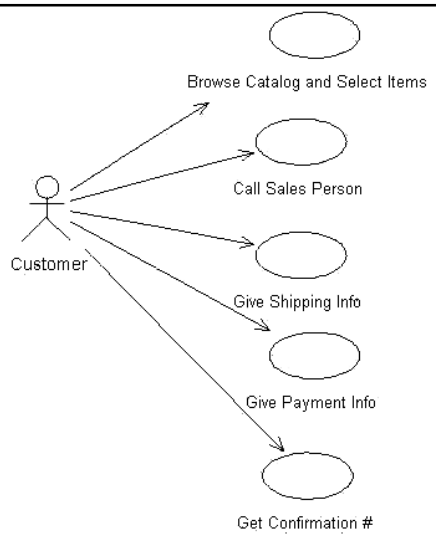
Start by listing a sequence of steps a user might take in order to complete an action.

Example: A user placing an order with a sales company might follow these steps.

1. Browse catalog and select items.
2. Call sales representative.
3. Supply shipping information.
4. Supply payment information.
5. Receive conformation number from salesperson.

These steps would generate this simple use case diagram.

Figure 7.3: Steps for Simple Use Case Diagram



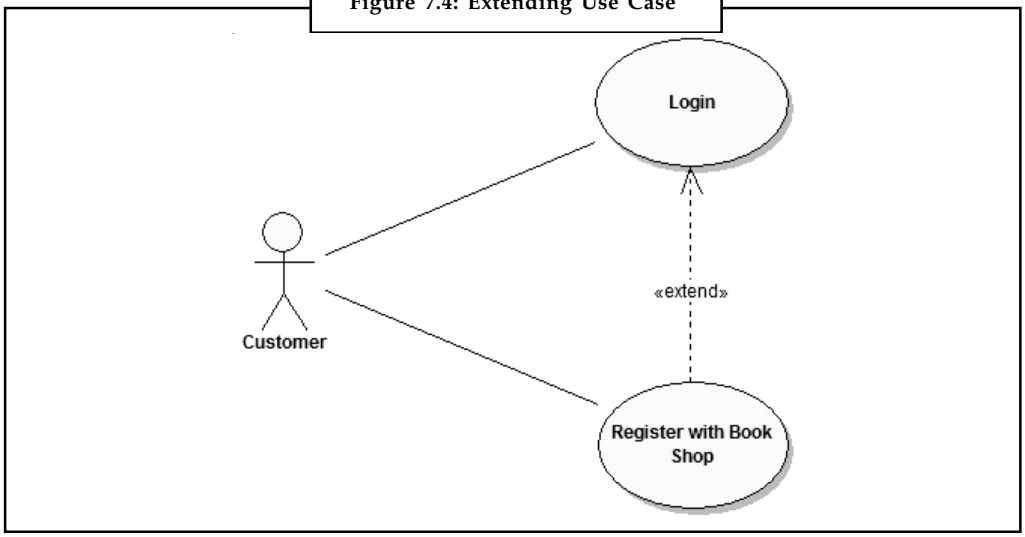
Notes

Example: The customer as an actor because the customer is using the ordering system. The diagram takes the simple steps listed above and shows them as actions the customer might perform. The salesperson could also be included in this use case diagram because the salesperson is also interacting with the ordering system.

From this simple diagram the requirements of the ordering system can easily be derived. The system will need to be able to perform actions for all of the use cases listed. As the project progresses other use cases might appear. The customer might have a need to add an item to an order that has already been placed. This diagram can easily be expanded until a complete description of the ordering system is derived capturing all of the requirements that the system will need to perform.

Each Use Case describes the functionality to be built in the proposed system, which can include another Use Case's functionality or extend another Use Case with its own behavior.

Figure 7.4: Extending Use Case



A Use Case description will generally includes:

- General comments and notes describing the use case.
- ***Requirements:*** The formal functional requirements of things that a Use Case must provide to the end user, such as <ability to update order>. These correspond to the functional specifications found in structured methodologies, and form a contract that the Use Case performs some action or provides some value to the system.
- ***Constraints:*** The formal rules and limitations a Use Case operates under, defining what can and cannot be done. These include:
  - Pre-conditions that must have already occurred or be in place before the use case is run.

    Example: <create order> must precede <modify order>
  - Post-conditions that must be true once the Use Case is complete.

    Example: <order is modified and consistent>
  - Invariants that must always be true throughout the time the Use Case operates.

Notes

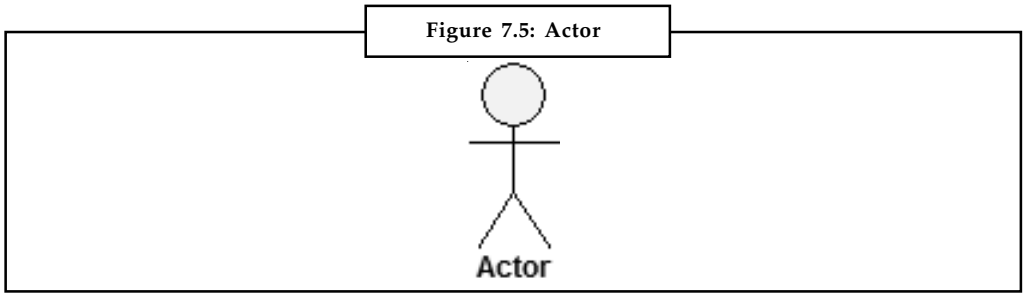
Example: An order must always have a customer number.

- ***Scenarios:*** Formal, sequential descriptions of the steps taken to carry out the use case, or the flow of events that occur during a Use Case instance. These can include multiple scenarios, to cater for exceptional circumstances and alternative processing paths. These are usually created in text and correspond to a textual representation of the Sequence Diagram.
- ***Scenario diagrams:*** Sequence diagrams to depict the workflow; similar to Scenarios but graphically portrayed.
- Additional attributes, such as implementation phase, version number, complexity rating, stereotype and status.

Actors

Use Cases are typically related to 'actors', which are human or machine entities that use or interact with the system to perform a piece of meaningful work that helps them to achieve a goal. The set of Use Cases an actor has access to define their overall role in the system and the scope of their action.

Figure 7.5: Actor



Self Assessment

Fill in the blanks:

1. A ..... model is a model of how different types of users interact with the system to solve a problem.
2. A ..... is an instance of a use case, and represents a single path through the use case.
3. ..... are used to describe the relationships between actors and the use cases they participate in.
4. ..... are typically related to 'actors', which are human or machine entities that use or interact with the system.
5. ..... is used to include optional behavior from an extending use case in an extended use case.

7.2 Sequence Models

Sequence diagram is the way to explain the behaviour of a system with a view to make interaction between the system and its surrounded environment. This interaction is arranged in a proper time sequence.

Sequence diagrams illustrate how objects interact with each other.

Notes

- They focus on message sequences, that is, how messages are sent and received among number of objects.
- Sequence diagrams have two axes: the vertical axis shows time and the horizontal axis shows a set of objects.
- The instance form describes a specific scenario in detail.
- The Generic form describes all possible alternatives in a scenario, therefore branches, conditions, and loops may be included.

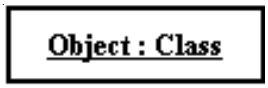
Sequence diagrams describe interactions among classes in terms of an exchange of messages over time.

Notes Historically, Charles Babbage was the first person to draw a diagram where each part of a machine was shown as a vertical line, time flows down the page, and connections between the parts flow across the page. This idea has been reinvented several times in the last 10 years.

7.2.1 Basic Sequence Diagram Symbols and Notations

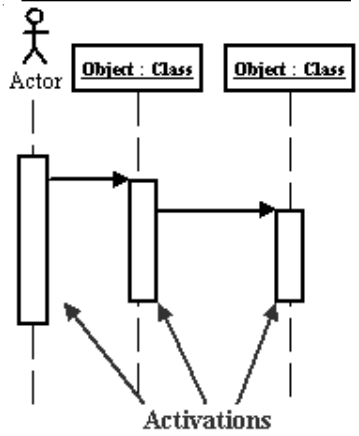
Class roles: Class roles describe the way an object will behave in context. Use the UML object symbol to illustrate class roles, but don't list object attributes.

Figure 7.6: Class Roles



Activation: Activation boxes represent the time an object needs to complete a task.

Figure 7.7: Activations



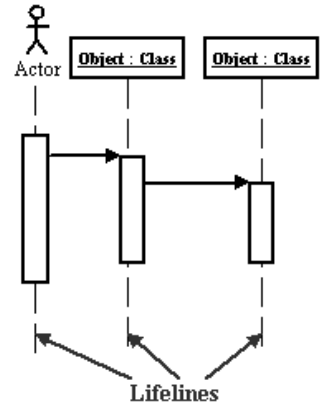
Messages: Messages are arrows that represent communication between objects. Use half-arrowed lines to represent asynchronous messages.

Caution Asynchronous messages are sent from an object that will not wait for a response from the receiver before continuing its tasks.

Notes

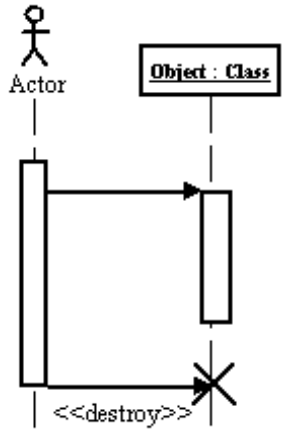
Lifelines: Lifelines are vertical dashed lines that indicate the object's presence over time.

Figure 7.8: Lifelines



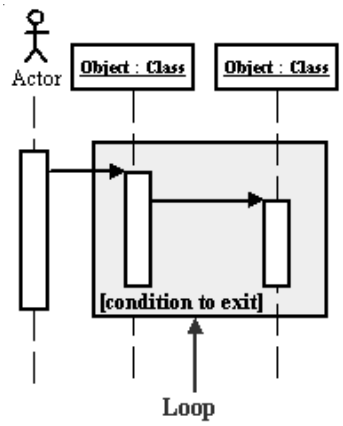
Destroying Objects: Objects can be terminated early using an arrow labeled "<< destroy >>" that points to an X.

Figure 7.9: Destroying Objects



Loops: A repetition or loop within a sequence diagram is depicted as a rectangle. Place the condition for exiting the loop at the bottom left corner in square brackets [ ].

Figure 7.10: Loops



Notes

Example: Sequence diagram are shown below.

Figure 7.11: The Concepts used in a Sequence Diagram

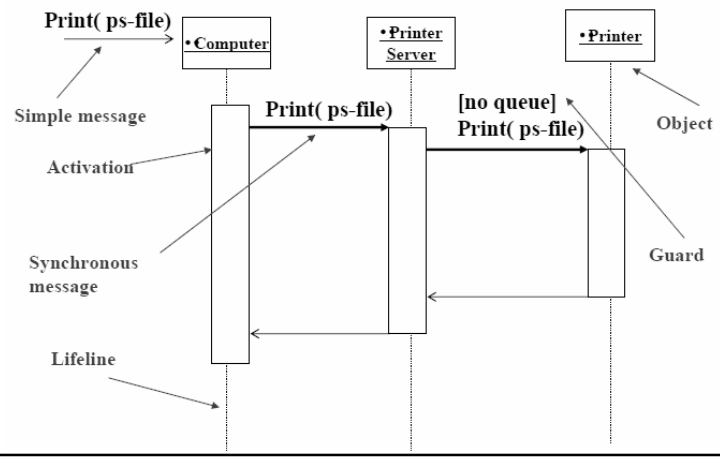


Figure 7.12: Sequence Diagram with Branch

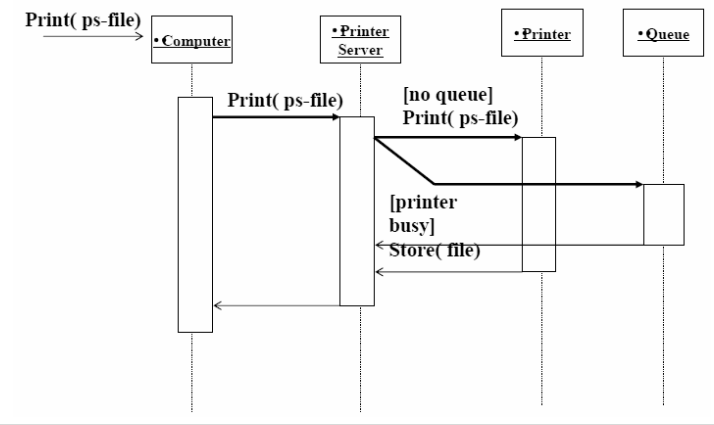
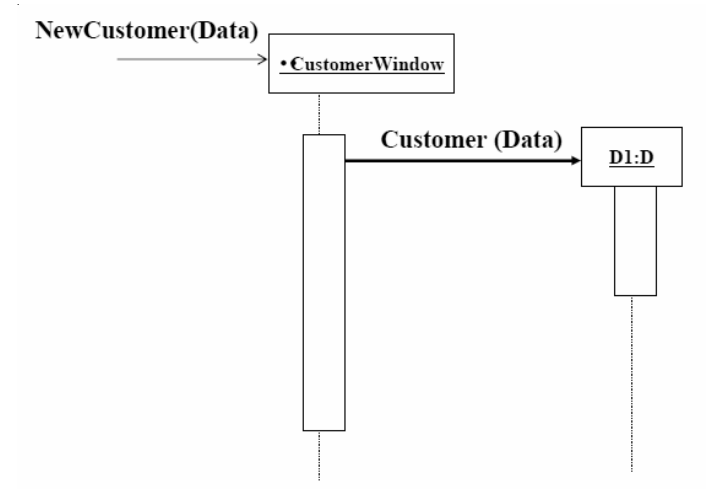


Figure 7.13: Creating Object



Notes

Figure 7.14: Destroying Object

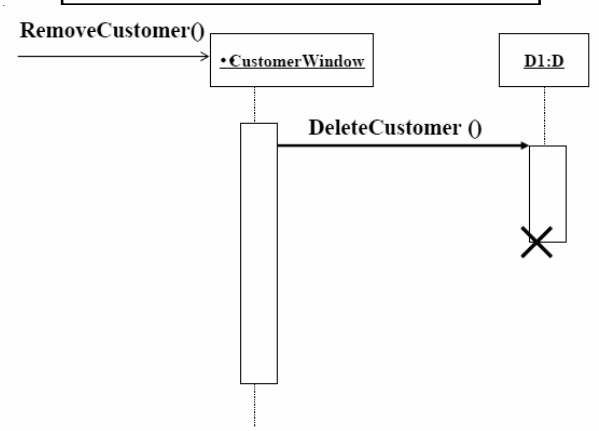
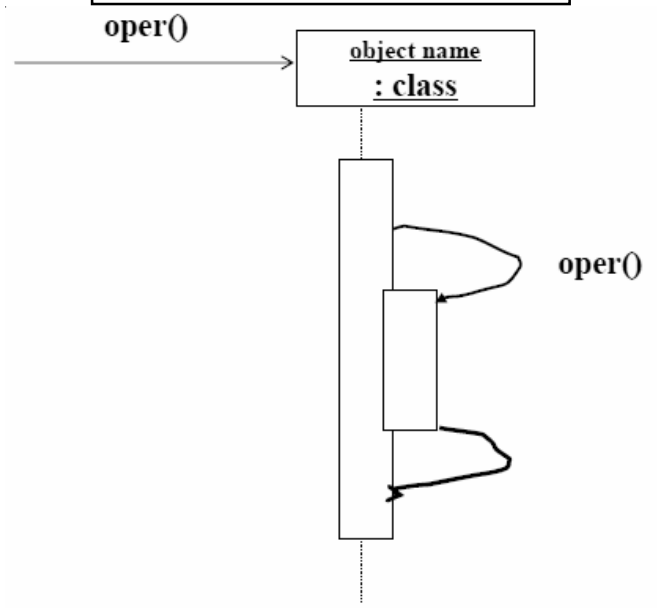


Figure 7.15: Recursion



Self Assessment

Fill in the blanks:

6. ..... diagrams illustrate how objects interact with each other.
7. ..... describe the way an object will behave in context.
8. ..... boxes represent the time an object needs to complete a task.
9. ..... are arrows that represent communication between objects.
10. ..... are vertical dashed lines that indicate the object's presence over time.

7.3 Activity Models

Activity diagram is a variation or special case of a state machine, in which the states are activities representing the performance of operations and the transitions are triggered by the completion

Notes

of the operations. Therefore activity diagram is a special kind of Statechart diagram, but showing the flow from activity to activity (not from state to state).

Activity – an ongoing non-atomic execution within a state machine. Activities ultimately result in some action. It's a real world process or execution of a software routine.

Action – made up of executable atomic computations that result in a change in state of the system or the return of a value (i.e., calling another operation, sending a signal, creating or destroying an object, or some pure computation).

Activity diagrams commonly contain:

- Activity states and action states
- Transitions
- Objects

Action states - executable, atomic computations (states of the system, each representing the execution of an action) – cannot be decomposed.

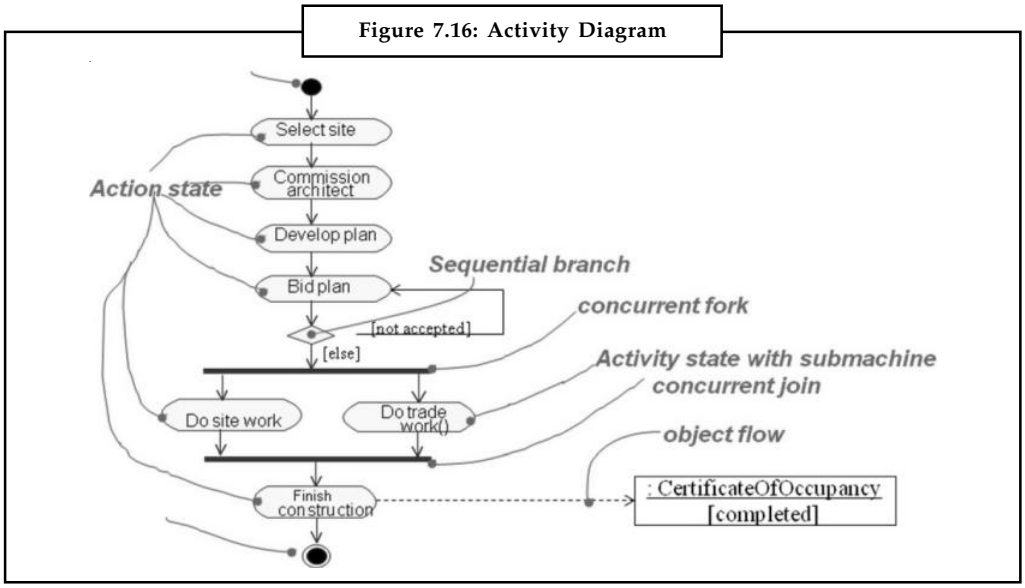
Activity states – non-atomic; can be further decomposed; can be represented by other activity diagrams – a composite whose flow of control is made up of other activity states and action states

An activity diagram illustrates the dynamic nature of a system by modeling the flow of control from activity to activity. An activity represents an operation on some class in the system that results in a change in the state of the system.

Did u know? Typically, activity diagrams are used to model workflow or business processes and internal operation.

Because an activity diagram is a special kind of statechart diagram, it uses some of the same modeling conventions.

Figure 7.16: Activity Diagram



7.3.1 Basic Activity Diagram Symbols and Notations

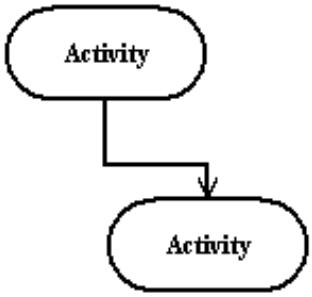
Action states: Action states represent the non-interruptible actions of objects. You can draw an action state in SmartDraw using a rectangle with rounded corners.

Notes

Figure 7.17: Action State

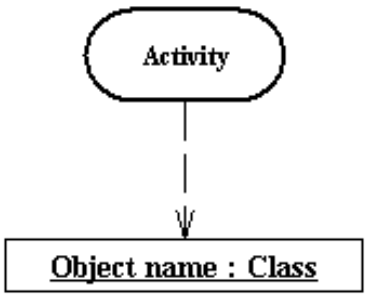
Action Flow: Action flow arrows illustrate the relationships among action states.

Figure 7.18: Action Flow



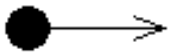
Object Flow: Object flow refers to the creation and modification of objects by activities. An object flow arrow from an action to an object means that the action creates or influences the object. An object flow arrow from an object to an action indicates that the action state uses the object.

Figure 7.19: Object Flow



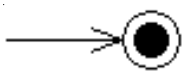
Initial State: A filled circle followed by an arrow represents the initial action state.

Figure 7.20: Initial State



Final State: An arrow pointing to a filled circle nested inside another circle represents the final action state.

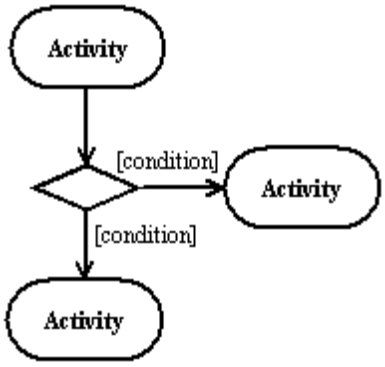
Figure 7.21: Final State



Branching: A diamond represents a decision with alternate paths. The outgoing alternates should be labeled with a condition or guard expression. You can also label one of the paths "else."

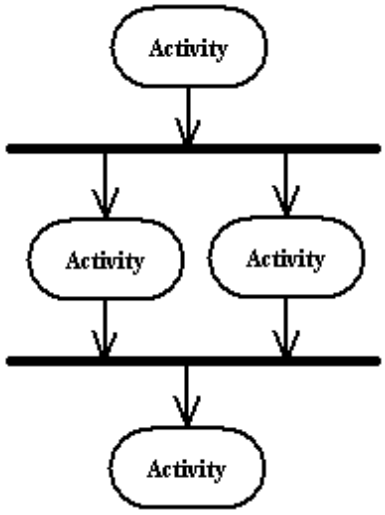
Notes

Figure 7.22: Branching



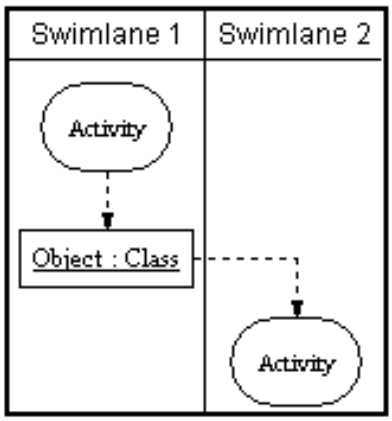
Synchronization: A synchronization bar helps illustrate parallel transitions. Synchronization is also called forking and joining.

Figure 7.23: Synchronization



Swimlanes: Swimlanes group related activities into one column.

Figure 7.24: Swimlanes



Notes

Task Compare and contrast object flow and action flow.

Caselet

Activity Diagram for Generating Restaurant Bill

Here, we will prepare an activity diagram for computing a restaurant bill. There should be a charge for each delivered item. The total amount should be subject to tax and service charge of 18% for groups of six or more. Any coupons charge submitted by the customer is subtracted from bill.

Source: http://www.programsformca.com/2012/03/activity-diagram-for-restaurant-exam.html

Self Assessment

Fill in the blanks:

11. An ..... diagram illustrates the dynamic nature of a system by modeling the flow of control from activity to activity.
12. ..... states represent the non-interruptible actions of objects.
13. ..... refers to the creation and modification of objects by activities.
14. A filled circle followed by an arrow represents the ..... action state.
15. ..... bar helps illustrate parallel transitions.

7.4 Summary

- The purpose of Interaction diagrams is to model interactions between objects, assist in understanding how a system (a use case) actually works, verify that a use case description can be supported by the existing classes Identify responsibilities/operations and assign them to classes.
- A use case model is a model of how different types of users interact with the system to solve a problem.
- A use case diagram is used to graphically depict a subset of the model to simplify communications.
- An actor represents a user or another system that will interact with the system you are modeling.
- Use Cases are typically related to 'actors', which are human or machine entities that use or interact with the system to perform a piece of meaningful work that helps them to achieve a goal.
- Sequence diagram is the way to explain the behaviour of a system with a view to make interaction between the system and its surrounded environment.
- Activity diagram is a variation or special case of a state machine, in which the states are activities representing the performance of operations and the transitions are triggered by the completion of the operations.

Notes

- An activity represents an operation on some class in the system that results in a change in the state of the system.

7.5 Keywords

Action flow: Action flow arrows illustrate the relationships among action states.

Active object: An object running under its own thread.

Activity diagram: Activity diagram is a variation or special case of a state machine, in which the states are activities representing the performance of operations and the transitions are triggered by the completion of the operations.

Attribute: A significant piece of data owned by a Class, often containing values describing each instance of the class. Besides the attribute name and a slot for the attribute value, an attribute may have specified Visibility, Type, Multiplicity, Default value, and Property-string.

Scenario: A scenario is an instance of a use case, and represents a single path through the use case.

Sequence diagram: Sequence diagram is the way to explain the behaviour of a system with a view to make interaction between the system and its surrounded environment.

Use case diagram: A use case diagram displays the relationship among actors and use cases.

Use case model: A use case model is a model of how different types of users interact with the system to solve a problem.

7.6 Review Questions

1. What are interaction diagrams? Analyze the purpose of interaction diagrams.
2. What is use case model? Discuss.
3. Elucidate the use case concept and use case diagrams with example.
4. Give explanation on the concept of use case package.
5. "Use cases are related to actors." Comment.
6. Discuss the constraints included in use case.
7. Illustrate the concept of sequence diagram with example.
8. Discuss the concept of loops within a sequence diagram. Illustrate with diagram.
9. What is an activity diagram? Discuss the basic activity diagram symbols and notations.
10. Make distinction between sequence diagram and activity diagram.

Answers: Self Assessment

1.	use-case	2.	Scenario
3.	Associations	4.	Use Cases
5.	<<extend>>	6.	Sequence
7.	Class roles	8.	Activation
9.	Messages	10.	Lifelines

Notes

11. activity
12. Action
13. Object flow
14. Initial
15. Synchronization

7.7 Further Readings

Books

Booch, Grady, (1994), *Object-oriented Analysis & Design,* Addison Wesley

Laganiere, Robert, (2004), *Object-oriented Software Engineering*, TMH

Rumbaugh, J., (2007), *Object-oriented Modelling and Design with UML,* Pearson Education

Satzinger, (2007), *Object-oriented Analysis & Design with the Unified Process,* Thomson

Online links

http://cit.snru.ac.th/UserFiles/OOAD_Unit08.pdf

http://debian.fmi.uni-sofia.bg/~vls/public/dsmt/uml/Lectures2/ooad9.pdf

http://www.scribd.com/doc/47759446/Lecture-9-OOAD

http://www.slideshare.net/jsm1979/ooad-with-uml-interaction-diagramming

Notes

Unit 8: Advanced Interaction Modelling

CONTENTS

Objectives

Introduction

8.1 Use Case Relationships

8.1.1 Communicates

8.1.2 Extends

8.1.3 Include or Uses

8.1.4 Generalization

8.2 Procedural Sequence Models

8.2.1 Sequence Diagrams with Passive Objects

8.2.2 Sequence Diagrams with Transient Objects

8.2.3 Guidelines for Procedural Sequence Models

8.3 Special Constructs for Activity Models

8.3.1 Conditional Threads

8.3.2 Nested Activity Diagram

8.4 Summary

8.5 Keywords

8.6 Review Questions

8.7 Further Readings

Objectives

After studying this unit, you will be able to:

- Define use case relationships
- Discuss the concept of sequence diagrams with passive objects
- Describe the concept of sequence diagrams with transient objects
- Explain the special constructs for activity models

Introduction

Interactions can be modeled at different levels of abstraction. At a higher level use cases describe how a system interacts with outside actors. Each use case represents a piece of functionality that a system provides to its users. Use cases are helpful for capturing informal software requirements. Sequence diagrams provide more details and show the messages exchanged among a set of object over time. Message includes both asynchronous signals and procedure call. They are good for showing the behavior sequences seen by users of a system. Finally, activity diagrams provide further details and show the flow of control among the steps of computation. In this

Notes

unit, we will discuss use case relationships, procedural sequence model, and special constructs for activity models.

8.1 Use Case Relationships

Use cases share different kinds of relationships. A relationship between two use cases is basically a dependency between the two use cases. Defining the relationship between two use cases is the decision of the modeler of the use case diagram.

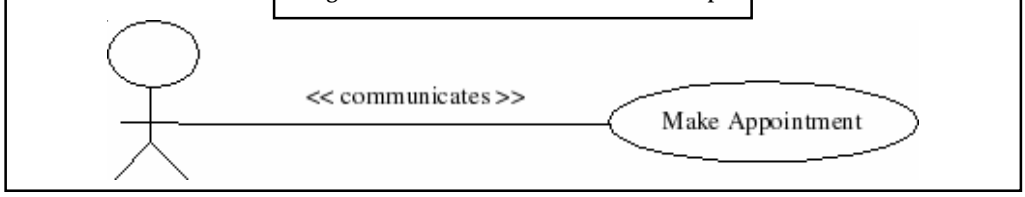
Did u know? The use of an existing use case using different types of relationships reduces the overall effort required in defining use cases in a system.

Use case relationships can be one of the following:

8.1.1 Communicates

The participation of an actor in a use case is shown by connecting the actor symbol to use case symbol by a solid path. The actor is said to 'communicate' with the use case. This is only relation between an actor and use cases. See figure 8.1.

Figure 8.1: Communicates Relationship



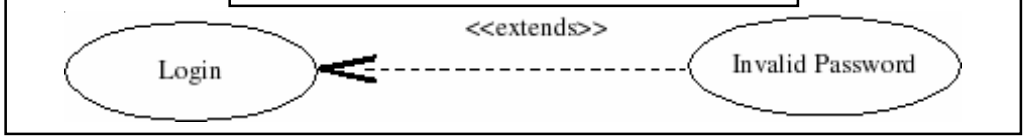
Source: http://gyan.frcrce.ac.in/~surve/OOAD/UCD/UCD_Relationships.html

8.1.2 Extends

An 'extends' shows the relationships between use cases. Relationship between use case A and use case B indicates that an instance of use case B may include (subject to specified in the extension) the behavior specified by A. An 'extends' relationship between use cases is depicted with a directed arrow having a dotted shaft. The tip of arrowhead points to the parent use case and the child use case is connected at the base of the arrow. The stereotype "<extends>" identifies as an extend relationship.

For example, validating the user for a system. An invalid password is extension of validating password use case as shown in figure 8.2.

Figure 8.2: An Example of an Extend Relationship



Source: http://gyan.frcrce.ac.in/~surve/OOAD/UCD/UCD_Relationships.html

8.1.3 Include or Uses

When a use case is depicted as using functionality of another functionality of another use case, this relationship between the use cases is named as an include or uses relationship. A uses

Notes

relationship from use case A to use case B indicates that an instance of the use case will also include the behavior as specified by B. An include relationship is depicted with a directed arrow having a dotted shaft. The tip of arrowhead points to the child use case and the parent use case connected at base of the arrow. The stereotype "<include>" identifies the relationship as an include relationship.

Example: The following figure illustrates an e-commerce application that provides customers with the option of checking the status of their orders. This behavior is modeled with a base use case called CheckOrderStatus that has an inclusion use case called LogIn. The LogIn use case is a separate inclusion use case because it contains behaviors that several other use cases in the system use. An include relationship points from the CheckOrderStatus use case to the LogIn use case to indicate that the CheckOrderStatus use case always includes the behaviors in the LogIn use case.

Figure 8.3: An Example of an include Relationship

Source:http://publib.boulder.ibm.com/infocenter/rsdvhelp/v6r0m1/index.jsp?topic=%2Fcom.ibm.xtools.modeler.doc%2Ftopics%2Fcinclude.html

The system boundary is potentially the entire system as defined in the requirements document. For large and complex systems, each module may be the system boundary.

Example: For an ERP system for an organization, each of the modules such as personal, payroll, accounting, etc. can form a system boundary for use cases specific to each of these business functions. The entire system can span all of these modules depicting the overall system boundary.

Notes In other words, in an include relationship; a use case includes the functionality described in the other use case as a part of its business process flow.

8.1.4 Generalization

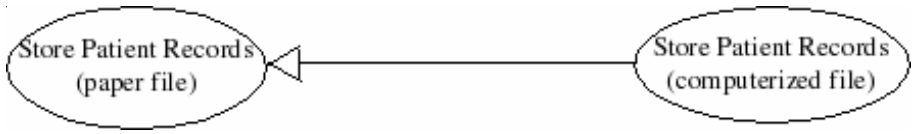
A generalization relationship is also a parent-child relationship between use cases. The child use case in the generalization relationship has the underlying business process meaning, but is an enhancement of the parent use case. In a use case diagram, generalization is shown as a directed arrow with a triangle arrowhead. The child use case is connected at the base of the arrow. The tip of the arrow is connected to the parent use case.

On the face of it, both generalization and extends appear to be more or less similar. But there is a subtle difference between a generalization and an extend relationship. When a generalization relationship is established between use cases, this implies that the parent use case can be replaced by the child use case without breaking the business flow. On the other hand, an extend relationship between use cases implies that the child use case enhances the functionality of the parent use case into a specialized functionality.

Notes

Caution The parent use case in an extend relationship can't be replaced by the child use case.

Figure 8.4: An example of Generalization Relationship



Source: http://gyan.frcrce.ac.in/~surve/OOAD/UCD/UCD_Relationships.html

From the diagram of a generalization relationship (Figure 8.4), you can that "Store_patient_records (paper file)" (parent) use case is depicted as a generalized version of the "Store_patient_records (computerized file)" (child) use case. Defining a generalization relationship between two implies that any occurrence of "Store_patient_records (paper file)" use case in the business flow of the system can be replaced with the "Store_patient_records (computerized file)" use case without impacting any business flow. This means that in future you might choose to store patient records in a computerized file instead of as paper documents without impacting other business actions.

Example: Consider example of Invalid Password use case which is extension of Login use case. The Login use case can't be replaced by Invalid Password use case. If you try to do this, you would not be able to seamlessly replace the occurrence of the "Login" use case with "Invalid Password" use case.

Task Make distinction between 'communicates' and 'generalization' relationship.

Notes In general, a child revises behavior subsequences at several different points of the parent sequence.

Self Assessment

Fill in the blanks:

1. Defining the relationship between two use cases is the decision of the ..... of the use case diagram.
2. A ..... between two use cases is basically a dependency between the two use cases.
3. ..... is only relation between an actor and use cases.
4. A '.....' relationship between use cases is depicted with a directed arrow having a dotted shaft.
5. In an ..... relationship, a use case includes the functionality described in another use case as a part of its business process flow.

Notes

6. The system boundary is potentially the entire system as defined in the ..... document.
7. In a use case diagram, generalization is shown as a directed arrow with a ..... arrowhead.
8. The parent use case in an extend relationship can't be replaced by the ..... use case.

8.2 Procedural Sequence Models

In the previous unit, we saw sequence diagrams comprising independent objects all of which are active at the same time. An object remains active after sending a message and can reply to other messages without waiting for a response. This is suitable for high level models.

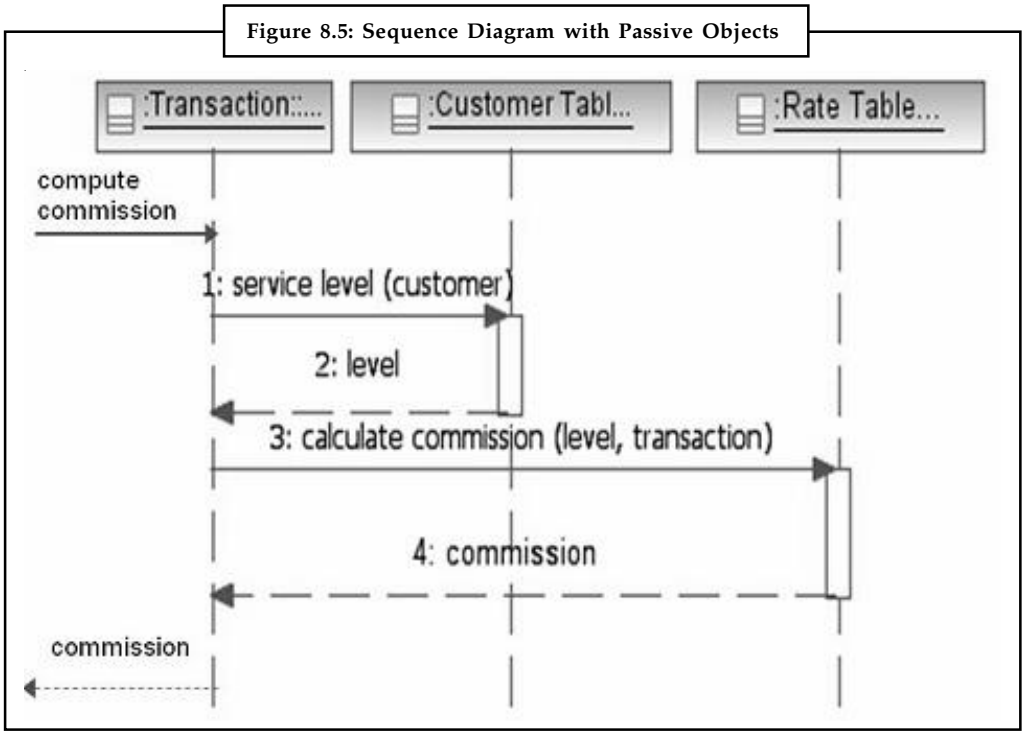
Did u know? Many of the implementations are procedural and restrict the number of objects that can execute at a time.

8.2.1 Sequence Diagrams with Passive Objects

Most objects are passive and do not have their own threads of control. Activation shows the time period during which a call of a method is being processed, including the time when the called method has invoked another operation.

Example: Figure 8.5 calculates the commission for the stock-brokerage transaction. The object of transaction obtains a request to calculate its commission. It receives the customer's service level from the customer table. Then it asks the rate table to calculate commission according to the service level, after which the commission value is returned to the caller.

Figure 8.5: Sequence Diagram with Passive Objects



Source:http://elearning.vtu.ac.in/13/ENotes/OOAD/Object %20Oriented%20Modeling%20and %20Design%20Patterns-%20Lecture%20%20%20Notes-Dr.pdf

Notes

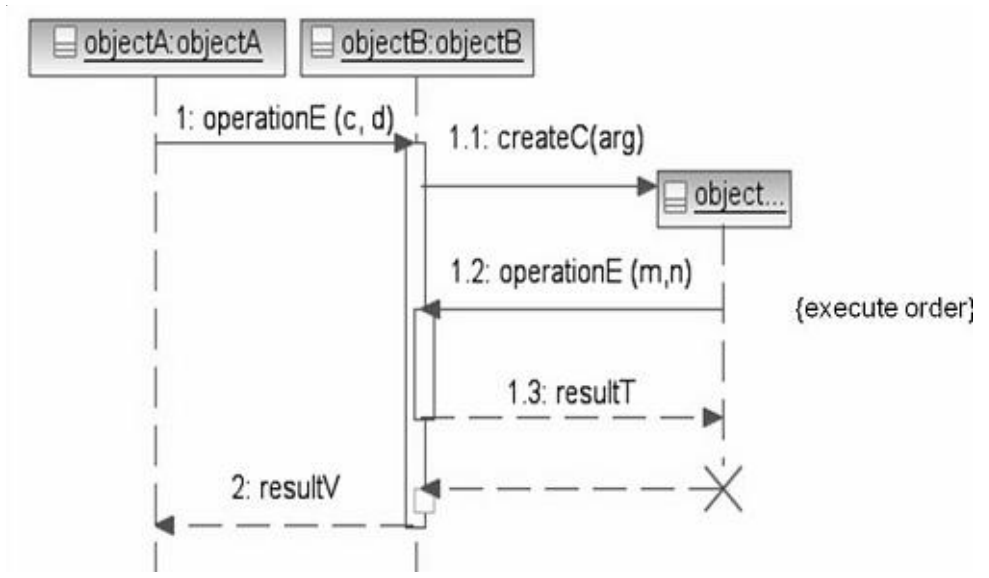
8.2.2 Sequence Diagrams with Transient Objects

ObjectA is an active object that initiates an operation. The notation for a call is an arrow from the calling activation created by the call. Activation, therefore, has a call arrow coming into its top and a return arrow leaving its bottom. If an object does not exist at the beginning of a sequence diagram, then it must be created during the sequence diagram.

Caution The UML shows creation by placing the object symbol at the head of the arrow for the call that creates the object.

Conditionals on a sequence diagram also can be shown.

Figure 8.6: Sequence Diagram with a Transient Object



Source:http://elearning.vtu.ac.in/13/ENotes/OOAD/Object%20Oriented%20Modeling%20and%20Design%20Patterns-%20Lecture%20%20%20Notes-Dr.pdf

8.2.3 Guidelines for Procedural Sequence Models

- Active vs. passive objects.

By definition, active objects are always activated and have their own focus of control.

- Advanced features.

Only show implementation details for difficult or especially important sequence diagrams.

Task Compare and contrast passive objects and active objects.

Notes

Self Assessment

Fill in the blanks:

9. An object remains active after sending a ..... and can reply to other messages without waiting for a response.
10. ..... objects do not have their own threads of control.
11. An ..... object owns a thread of control.
12. In sequence diagram with transient objects, ... has a call arrow coming into its top and a return arrow leaving its bottom.
13. In sequence diagram with passive objects, activation shows the ..... during which a call of a method is being processed.

8.3 Special Constructs for Activity Models

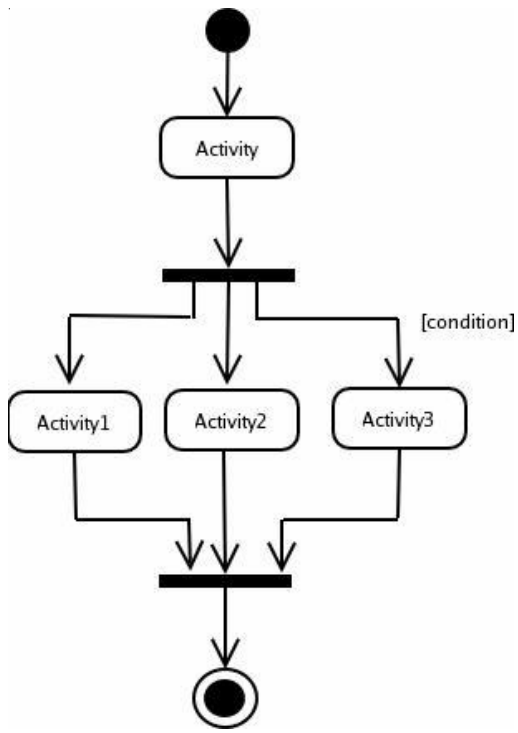
Activity models include some complex constructs which are discussed below:

- Conditional threads
- Nested activity diagrams

8.3.1 Conditional Threads

Guard conditions can be used to show that one of a set of concurrent threads is conditional.

Figure 8.7: Conditional Thread



Source: http://gyan.frcrce.ac.in/~surve/OOAD/UCD/UCD_Relationships.html

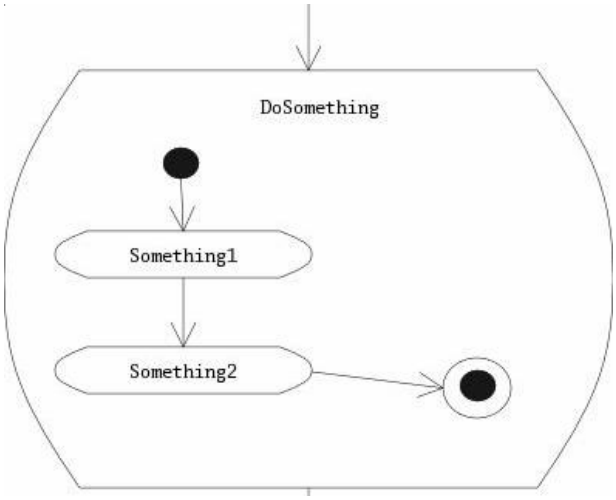
Notes

8.3.2 Nested Activity Diagram

An activity state may reference another activity diagram, which shows the internal structure of the activity state. Another way to say this is that you can have nested activity graphs.

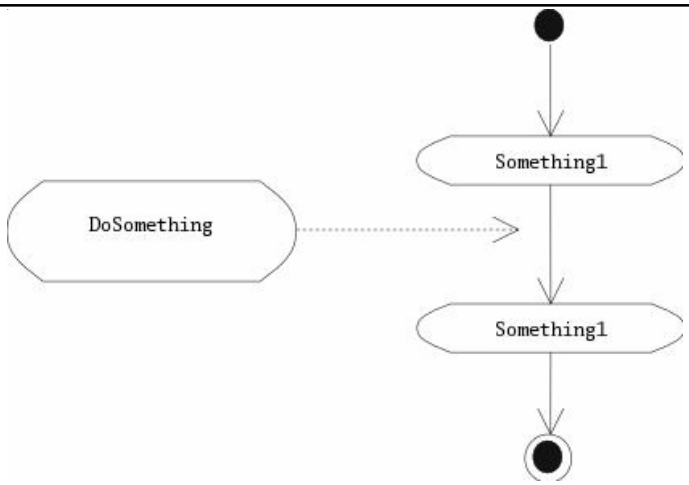
For example, you can either show the sub-graph inside of the activity state (Figure 8.8), or let the activity state refer to another diagram (Figure 8.9).

Figure 8.8: A Nested Activity Graph Shown Within an Activity State



Source: http://gyan.frcrce.ac.in/~surve/OOAD/UCD/UCD_Relationships.html

Figure 8.9: Alternative Activity State Referring to another Activity Diagram



Source: http://gyan.frcrce.ac.in/~surve/OOAD/UCD/UCD_Relationships.html

Self Assessment

Fill in the blanks:

14. ..... conditions can be used to show that one of a set of concurrent threads is conditional.

15. ..... graphs shows the internal structure of the activity state.

Notes

Case Study

Answering System

The Answering System is system foe answering phone calls and recording messages from callers. It is intended as a personal answering system for a single owner. It will support:

- Modes for announce only and accepting caller messages
- Ability to review caller messages
- Personalized greetings
- Local management of modes, greetings, and caller messages

Answering System Domain Use Case Model

Let us analyze the requirement document to identify the potential actors and use cases of the system. First, let's list the potential actors. A quick look at requirement document shows the following terms and entities specific to the system:

- The caller is the person who is answered and their messages are taken
- The owner is person who records the greetings, set the answer mode and reviews the caller me

Identifying Actor

There are certain terms and entities in the list that identify that they perform certain roles or business processes. From the preceding list, we can see that there are some entities that perform an action and some that form the target for the action. The entities that perform action will be the actor for Answering System. In the above list, the actor that we can identify are:

- Owner
- Caller

Identifying Use Cases

Next, let's identify the potential business processes in the Answering System. The primary business flows in the system are:

- Review Caller Messages
- Answer Caller
- Set Answer Mode
- Record Greetings

As we analyze the requirement document further, we can determine some discrete processes within these primary business flows. To review caller messages, the owner needs to have ability to delete caller message. So, within the "Review Caller Messages" use case, we can identify following use case:

- Delete Caller Message

The "Answer Caller" use case can be refined into smaller discrete processes such as play greeting, take caller message. Now, the use cases that we identified within the "Answer Caller" are:

- Play Greeting
- Take Caller Message

Contd...

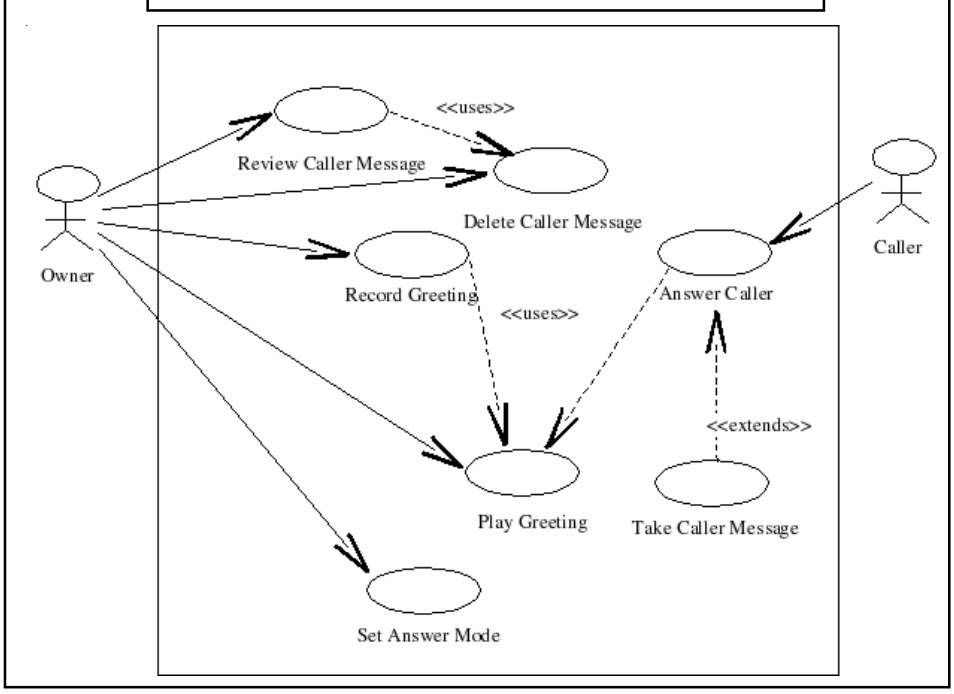
Notes

And similarly, "Record Greeting" use case uses the discrete process - play greeting. Our final list of use cases for Answering System will be:

- Review Caller Messages
- Answer Caller
- Set Answer Mode
- Record Greetings
- Delete Caller Message
- Play Greeting
- Take Caller Message

If you are analyzing a sentence in English, the subject in the sentence can be identified as a potential actor and the verb part of the sentence can be potential use case. Remember, this may or may not apply to the problem in hand, but is a good starting point for use case modeling.

Figure 1: The Use Case Diagram for the Answering System



A sample use case description is given in Table 1.

Table 1: Use Case Description for Use Case Answer Caller

Use Case ID	#CRCEAS-UC1
Use Case Type	Concrete
Use Case Name	Answer Caller
Basic Course	Actors: Caller Preconditions: Answering machine is idle Primary Path: • The caller rings the phone line connected to the answering machine.

Contd...

Notes

	• The system waits for the ring count to reach the answering ring count. • The system plays a greeting. • The system performs mode specific processing. Postconditions: The answering machine prepares for the next caller.
Alternative Course	• The caller hangs up phone during the playing of the greeting. • The system recognizes hang-up, stops greeting and prepares for the next caller.
Extensions	Take Caller Message
Decision Support	Frequency: This use is triggered on every incoming call when machine is set to answering mode.
Modification History	Owner: Project Group A, XYZ Ltd. Initiation Date: January 17, 2005. Date Last Modified: January 27, 2005.

Question

Illustrate the Use Case Diagram for the Answering System.

Source: http://gyan.frcrce.ac.in/~surve/OOAD/UCD/UCD_CaseStudy.html

8.4 Summary

- A relationship between two use cases is basically a dependency between the two use cases. Defining the relationship between two use cases is the decision of the modeler of the use case diagram.
- The participation of an actor in a use case is shown by connecting the actor symbol to use case symbol by a solid path. The actor is said to 'communicate' with the use case.
- An extends shows the relationships between use cases. Relationship between use case A and use case B indicates that an instance of use case B may include the behavior specified by A.
- When a use case is depicted as using functionality of another functionality of another use case, this relationship between the use cases is named as an include or uses relationship.
- A generalization relationship is a parent-child relationship between use cases. In a use case diagram, generalization is shown as a directed arrow with a triangle arrowhead.
- An object remains active after sending a message and can reply to other messages without waiting for a response.
- Guard conditions can be used to show that one of a set of concurrent threads is conditional.
- An activity state may reference another activity diagram, which shows the internal structure of the activity state.

8.5 Keywords

Active object: An active object owns a thread of control.

Communicates relationship: A communicates relationship between an actor and a use case indicates that the actor initiates the use case.

Extends: An extends shows the relationships between use cases.

Generalisation: A generalisation relationship between actors or use cases indicates that one actor or use case (the child) inherits the characteristics of another actor or use case (the parent).

Notes

Guard conditions: Guard conditions can be used to show that one of a set of concurrent threads is conditional.

Include relationship: An include relationship suggests that one use case must include another.

Nested activity graphs: Nested activity graphs show the internal structure of the activity state.

Passive object: A passive object does not have its own thread of control.

8.6 Review Questions

1. Discuss the concept of relationship between two use cases.
2. Which relationship is considered as the relation between an actor and use cases? Illustrate.
3. Explain the concept of 'extends' relationship with example.
4. Illustrate the use of 'include' relationship with example.
5. Which symbol is used to depict generalization? Illustrate with example.
6. Make distinction between 'include' relationship and 'generalisation' relationship.
7. Discuss the concept of procedural sequence models.
8. What are passive objects? Illustrate the sequence diagram with passive objects with example.
9. Discuss the guidelines used for procedural sequence models.
10. Elucidate the concept of nested activity diagram with example.

Answers: Self Assessment

1.	Modeler	2.	Relationship
3.	Communicates	4.	Extends
5.	include	6.	Requirements
7.	triangle	8.	Child
9.	message	10.	Passive
11.	active	12.	Activation
13.	time period	14.	Guard
15.	Nested activity		

8.7 Further Readings

Books

Booch, Grady, (1994), *Object-oriented Analysis & Design,* Addison Wesley

Laganiere, Robert, (2004), *Object-oriented Software Engineering,* TMH

Rumbaugh, J., (2007), *Object-oriented Modelling and Design with UML,* Pearson Education

Satzinger, (2007), *Object-oriented Analysis & Design with the Unified Process, Thomson*

Notes

Online links

http://cuiwww.unige.ch/isi/cours/UML-en/06-sequences.pdf

http://www.cs.ucy.ac.cy/~epl233/labs/EPL233-Lab7.pdf

http://www.karonaconsulting.com/downloads/UseCases_IncludesAndExtends.pdf

http://www.sparxsystems.com/resources/tutorial/use_case_model.html

Notes

Unit 9: Analysis and Design

CONTENTS

Objectives

Introduction

9.1 Object Oriented Analysis and Design Process Overview

9.2 Development Life Cycle

9.2.1 Approaches to Systems Testing

9.2.2 Object-oriented Approach: A Use Case Driven Approach

9.3 Summary

9.4 Keywords

9.5 Review Questions

9.6 Further Readings

Objectives

After studying this unit, you will be able to:

- Describe Analysis and Design Process Overview
- Discuss the various phases of development life cycle
- Explain the approaches to systems testing
- Define use case driven approach

Introduction

Object-oriented Analysis and Design (OOAD) is a software engineering approach that models a system as a group of interacting objects. Each object represents some entity of interest in the system being modeled, and is characterised by its class, its state (data elements), and its behaviour. Various models can be created to show the static structure, dynamic behaviour, and run-time deployment of these collaborating objects. There are a number of different notations for representing these models, such as the Unified Modeling Language (UML). In this unit, we will discuss various phases that take place in development life cycle.

9.1 Object Oriented Analysis and Design Process Overview

Object-oriented Analysis (OOA) applies object-modeling techniques to analyze the functional requirements for a system. Object-oriented Design (OOD) elaborates the analysis models to produce implementation specifications. OOA focuses on what the system does, OOD on how the system does it.

One can make a distinction between analysis and design. Analysis tends to be in the domain of the client, design tends to be in the domain of the developer, and their focus is different. But this distinction is more of a management one than a technological one.

Notes

All participants in the process need to be familiar with their context: where do their inputs come from and who uses their outputs. A business analyst operates in the requirements domain, but they must understand both the customer's needs and the task faced by the technical analyst who translates their requirements into the problem domain. Similarly, technical analysts must understand both requirements and the technical difficulties faced by developers who must work in the solution domain.

The requirement document will be the main document for developers, testers and database administrators. In other words, this is the main document that will be referred by everyone. After the requirement documents, other detailed documents many be needed.

Example: The architectural design which is a blueprint for the design with the necessary specifications for the hardware, software, people and data resources.

Caution Developers need to understand the problem domain model, and they have to understand the target technology.

The object-oriented methodology developed based on the lack of synergy between process-centered and data-centered approaches in SDLC (software development life cycle). Decomposition of system into a set of process (process centric) or data (data centric) cannot be easily obtained, as both aspects are closely related one another.

It is difficult to develop system by primarily focusing only to one aspect. As result, the system produced tends to be extendable only in one world. A process centric developed system cannot be easily extended when there are changes in type of data in the system. This kind of problems also exists in the data centric developed system.

OO methodology decomposes problems into objects. Objects are considered part of the system that contains both process and data, an object may do some activities/processes (mapped as object methods), an object may also have states (mapped as object attributes). This way, developers will focus on the entity in the system that actually does processes and carries data, rather than focus primarily only to one aspect.

Notes OO-based system development extensively uses a tool called UML (Unified Modeling Language), which is a set of standard in diagramming and modeling techniques invented by three OO champions, Grady Booch, Ivar Jacobson, and James Rumbaugh, when they worked together in Rational Software. In 1997 UML proposed to and accepted by the Object Management Group (OMG) as a standard diagramming notation in object-oriented system development.

An OO approaches in system development must be:

- ***Use case Drive:*** This means that use case is the primary modeling tool to define system behavior. Use cases describe how the users of the system interact with the system to perform activity. And as a use case focuses only to one activity at a time, it is inherently simple.
- ***Architecture Centric:*** The term architecture centric gives a high level view of the system being developed. The software architecture chosen for the system should drive the specification, construction, and documentation of the system itself. The system architecture must support three views of the system:

Notes

- ***Functional view:*** Describes system behavior from the perspective of users of the system. Use case diagrams used to depict this functional view.
- ***Static view:*** Describes the structure of the system in terms of classes, methods, attributes, and relationships of objects in the system. This view is depicted using CRC (Class Responsibility Collaboration) cards, as well using class and object diagrams.
- ***Dynamic view:*** Describes the internal system behavior in terms of object communications and change of states. UML tools used to depict this view are sequence diagrams, collaboration diagrams, and object state-charts.

- ***Iterative and Incremental:*** Iterative and Incremental paradigm means that each iteration of the system development must bring the system closer to the requirements. As SDLC is a gradual process, the UML diagrams used in OO-based development moves from a conceptual and abstract thing in the analysis and design phase to become more and more detail in the implementation phase.

Task Make distinction between static view and dynamic view.

Self Assessment

Fill in the blanks:

1. ..... applies object-modeling techniques to analyze the functional requirements for a system.
2. ..... elaborates the analysis models to produce implementation specifications.
3. The ..... document will be the main document for developers, testers and database administrators.
4. OO methodology decomposes problems into .....
5. ..... describe how the users of the system interact with the system to perform activity.
6. ..... view describes system behavior from the perspective of users of the system.
7. .....view describes the structure of the system in terms of classes, methods, attributes, and relationships of objects in the system.
8. ..... view describes the internal system behavior in terms of object communications and change of states.

9.2 Development Life Cycle

It is imperative to get acquainted with the general phases through which software development passes. Unlike consumer products, software is not manufactured. In this sense, software is not a passive entity rather it behaves organically. It undergoes a series of evolutionary stages throughout its lifetime – starting from a problem terminating into a solution. That is why software is said to 'develop' or 'evolve' and not manufactured.

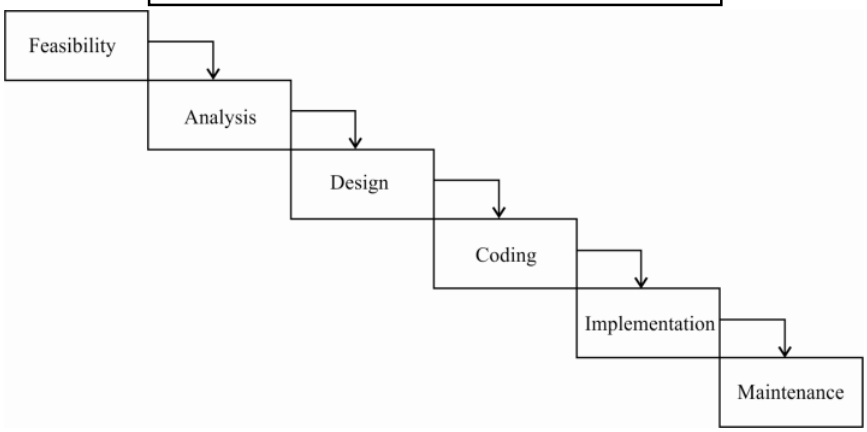
In other words, software is born, passes through various developmental phases, gets established, undergoes maintenance and finally grows old before being commissioned out of service.

Notes

Software engineers have developed a number of different 'life styles' through which software passes. In general these life styles are known as software development life cycle or SDLC in short.

Following are the developmental phases of software:

Figure 9.1: System Development Life Cycle Phases



1. ***Feasibility Study:*** The origin of any software begins with the study of the problems, which it intends to solve. Software cannot be envisaged unless there is a problem that it must solve. Therefore, studying the problem in depth, understanding the true nature of the problem and representing the problem in comprehensible manner is what necessitates inclusion of this phase. The feasibility study is used to determine if the project should get the go-ahead. If the project is to proceed, the feasibility study will produce a project plan and budget estimates for the future stages of development.

2. ***Analysis:*** It is a detailed study of the various operations performed by the proposed software. A key question that is considered in this phase of development is - What must be done to solve the problem? One aspect of analysis is defining the boundaries or interface of the software.

   During analysis, data are collected in available files, decision points, and transactions handled by the present system. Bias in data collection and interpretation can be fatal to the developmental efforts. Training, experience and common sense are required for collection of the information needed to do the analysis.

   Once analysis is completed the analyst has a firm understanding of what is to be done. The next step is to decide how the problem might be solved. Thus, in the software systems design, we move from the logical to the physical aspects of the life cycle.

3. ***Design:*** The most creative and challenging phase of software life cycle is design. The term design describes both a final software system and a process by which it is developed. It refers to the technical specifications (analogous to the engineer's blueprints) that will be applied in implementing the software system. It also includes testing the software. The key question around which this phase revolves is - How should the problem be solved?

   Generally, designers make two documents, namely **High Level Document** and **Low Level Document.**

Example: For making the design document of a building, High Level Document will contain information at a very higher level like how many floors and how many houses in each floor ?

Low Level Document contains information at a very low level.

Notes

Example: Detailed Design of each house.

Design phase is itself sub-divided into sub-phases discussed below:

The first step is to determine how the output is to be produced and in what format. Samples of the output (and input) are outlined.

Second, input data and master files (data base) have to be designed to generate the required output. The operational (processing) phase are handled through program construction and testing, including a list of the programs needed to meet the software objectives and complete documentation.

Finally, details related to justification of the system and an estimate of the impact of the software on the user are documented and evaluated before it is implemented.

4. ***Coding:*** In this phase the designs are translated into code. Computer programs are written using a conventional programming language or an application generator. Programming tools like Compilers, Interpreters, and Debuggers are helpful in generating and checking the code. Different high level programming languages like C, C++, Pascal, and Java are used for coding.

Did u know? With respect to the type of application, the right programming language is chosen.

5. ***Implementation:*** The final stage of initial development is implementation where the software is put into production and runs actual business.

   In this phase, the organization configures and enables system security features, tests the functionality of these features, installs or implements the system, and obtains a formal authorization to operate the system.

Caution Design reviews and system tests should be performed before placing the system into operation to ensure that it meets all required security specifications.

In addition, if new controls are added to the application or the support system, additional acceptance tests of those new controls must be performed. This approach ensures that new controls meet security specifications and do not conflict with or invalidate existing controls.

Notes The results of the design reviews and system tests should be fully documented, updated as new reviews or tests are performed, and maintained in the organization's official records.

6. ***Maintenance:*** Change is inevitable. Software serving the user's needs in the past may become less useful or sometimes useless in the changed environment. User's priorities, changes in organizational requirements, or environmental factors may call for software enhancements.

Notes

9.2.1 Approaches to Systems Testing

Test is done according to

- how it has been built
- what it should do

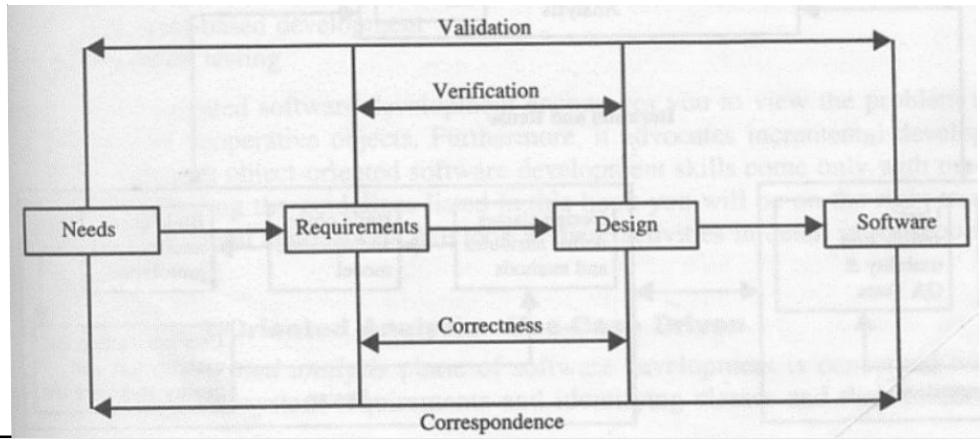
It includes four quality measures:

- ***Correspondence:*** It measures how well delivered system matches needs of operational environment, as described in original requirements statement
- ***Validation:*** It is defined as the task of predicting correspondence.

Did u know? True correspondence is only determined after system is in place.

- ***Correctness:*** It measures consistency of product requirements with respect to design specification
- ***Verification:*** It is the exercise of determining correctness (correctness objective => always possible to determine if product precisely satisfies requirements of specification)

Figure 9.2: Quality Measures for Software Evaluation



Source: http://www.b-u.ac.in/sde_book/object_orient.pdf

Task Compare and contrast validation and verification.

9.2.2 Object-oriented Approach: A Use Case Driven Approach

Object-oriented software development life cycle consists of

- Object-oriented analysis
- Object-oriented design
- Object-oriented implementation

Notes

Use case model can be employed throughout most activities of software development

- designs traceable across requirements, analysis, design, implementation & testing can be produced.
- all design decisions can be traced back directly to user requirements usage scenarios can be test scenarios.

Self Assessment

Fill in the blanks:

9. The ..... study is used to determine if the project should get the go-ahead.
10. Analysis is a ..... study of the various operations performed by the proposed software.
11. The term design describes both a final ..... system and a process by which it is developed.
12. In case of ....., the software is put into production and runs actual business.
13. ..... measures how well delivered system matches needs of operational environment, as described in original requirements statement.
14. ..... measures consistency of product requirements with respect to design specification.
15. ..... is the exercise of determining correctness.

9.3 Summary

- Object-oriented Analysis (OOA) applies object-modeling techniques to analyze the functional requirements for a system.
- Object-oriented Design (OOD) elaborates the analysis models to produce implementation specifications.
- The feasibility study is used to determine if the project should get the go-ahead.
- During analysis, data are collected in available files, decision points, and transactions handled by the present system.
- The term design describes both a final software system and a process by which it is developed.
- In coding phase the designs are translated into code. Computer programs are written using a conventional programming language or an application generator.
- The final stage of initial development is implementation where the software is put into production and runs actual business.
- Validation includes task of predicting correspondence (true correspondence only determined after system is in place.
- Verification includes exercise of determining correctness (correctness objective => always possible to determine if product precisely satisfies requirements of specification).

Notes

9.4 Keywords

Analysis: A detailed study of the various operations performed by the proposed software.

Design: The term design describes both a final software system and a process by which is developed.

Feasibility study: The feasibility study is used to determine if the project should get the go-ahead.

Implementation: This phase is primarily concerned with coding the software design into an appropriate programming language. Testing the programs and installing the software.

Maintenance: In this phase the software is continuously evaluated and modified to suit the changes as they occur.

Use Cases: Use cases describe how the users of the system interact with the system to perform activity.

Validation: Validation includes task of predicting correspondence (true correspondence only determined after system is in place.

Verification: Verification includes exercise of determining correctness (correctness objective => always possible to determine if product precisely satisfies requirements of specification).

9.5 Review Questions

1. Describe the process of object-oriented analysis and design.
2. What are the characteristic features of object-oriented analysis and design? Discuss.
3. Explain the concept of development life cycle.
4. Describe the various developmental phases of software. Illustrate with diagram.
5. Discuss various object oriented approaches used in system development.
6. "Design reviews and system tests should be performed before placing the system into operation." Comment.
7. What are the three views supported by the system architecture? Discuss.
8. Make distinction between analysis and design phases of the development life cycle.
9. Illustrate the sub-phases included in design phase.
10. Elucidate the quality measures used for software evaluation.

Answers: Self Assessment

1.	Object-oriented Analysis (OOA)	2.	Object-oriented Design (OOD)
3.	requirement	4.	Objects
5.	Use-cases	6.	Functional
7.	Static	8.	Dynamic
9.	feasibility	10.	Detailed
11.	software	12.	Implementation

13. Correspondence
14. Correctness
15. Verification

Notes

9.6 Further Readings

Books

Booch, Grady, (1994), *Object-oriented Analysis & Design,* Addison Wesley

Laganiere, Robert, (2004), *Object-oriented Software Engineering,* TMH

Rumbaugh, J. (2007), *Object-oriented Modelling and Design with UML,* Pearson Education

Satzinger, (2007), *Object-oriented Analysis & Design with the Unified Process,* Thomson

Online links

http://www.freepatentsonline.com/article/International-Journal-Business-Research/190463129.html

http://www.umsl.edu/~sauterv/analysis/488_f01_papers/quillin.htm

http://www.bvicam.ac.in/general/courseMaterial/MCAIV/OOAD/Lecture%20Notes/UNIT-I.pdf

http://ooadclick.blogspot.in/2011/11/software-development-life-cyclesdlc.html

Notes

Unit 10: System Conception

CONTENTS

Objectives

Introduction

10.1 Devising and Elaboration

10.1.1 Devising a System Concept

10.1.2 Elaboration

10.2 Preparing a Problem Statement

10.2.1 Issues in Complexity Modelling and Abstraction

10.3 Summary

10.4 Keywords

10.5 Review Questions

10.6 Further Readings

Objectives

After studying this unit, you will be able to:

- Describe how to devise a system concept
- Explain the concept of elaboration
- Discuss the concept of preparing a problem statement
- Analyse the issues in complexity modelling and abstraction

Introduction

System conception deals with the genesis of an application. Primarily some person, who understand technology as well as business needs, thinks about an idea for an application. The intention of system conception is to recognize the big picture, that is, what requirements does the proposed system meet, can it be developed at a reasonable cost?, etc. System conception is considered as a requirement analysis phase. Requirements describe how system behaves from the user's point of view. True customer requirements should be separated from design decisions. Solution should be deferred until a problem is fully understood. Requirements statements are typically ambiguous, incomplete and inconsistent. Some of requirements are plain wrong, some impose unreasonable implementation cost or may create new problems, if implemented.

10.1 Devising and Elaboration

In this section, we will discuss devising a system concept and the concept of elaboration.

10.1.1 Devising a System Concept

Some ways to find new system concepts:

- ***New functionality:*** add functionality to existing system

Notes

- ***Streamlining:*** remove restriction the way system work
- ***Simplification:*** let ordinary person perform task
- ***Automation:*** automate manual process
- ***Integration:*** combine functionality from different system
- ***Globalisation:*** travel to other country & observe their cultural & business practice

Task Make distinction between automation and integration.

10.1.2 Elaboration

Elaboration is the point where you want to have a better understanding of the problem:

- ***Who is the application for?***
  - understand which person & organization are stakeholder (financial sponsors & end user) for new system
  - financial sponsors are important because they are paying for system & expect the project to be success & within budget
  - user will determine success of system
- ***What problem will it solve?***
  - determine which feature will be in new system & which will not
- ***Where it will be used?***
  - determine new system is complement of existing system, it will be used locally or distributed via a network or it just a new capability that you deploy without disrupting workflow
- ***When is it needed?***
  - *Feasible time:* the time in which system can be developed within cost & resources
  - *Required time:* when system needed to meet business goals
- ***Why it is needed?***
  - understand motivation for new system
  - business case will give you insight what stakeholders expect
  - How it will work?

The purpose of elaboration is to analyse the problem, develop the project plan further, and eliminate the riskier areas of the project. By the end of the elaboration phase, we aim to have a general understanding of the entire project, even if it is not necessarily a deep understanding (that comes later, and in small, manageable chunks).

Two of the UML models are often invaluable at this stage. The Use Case Model helps us to understand the customer's requirements, and we can also use the Class Diagram to explore the major concepts our customer understands.

In the Elaboration Phase, we are concerned with exploring the problem in detail, understanding the customer's requirements and their business, and to develop the plan further.

Notes

We must get in to the correct frame of mind to attack this phase correctly. We must try not to get bogged down with too much detail, especially implementation details.

Caution We need to have a very broad view of the system and understand system-wide issues.

Prototyping

A key activity in the Elaboration Phase is the mitigation of risks. The sooner risks are identified and shot down, the lesser their impact will be on the project. Prototyping difficult or problematic areas of the project are a tremendous help in the mitigation of risks. Given that we don't want to get bogged down in implementation and design at this phase, the prototypes should be very focused, and explore just the area of concern. Prototypes can be thrown away at the end of the exercise, or they can be reused during the construction phase.

Deliverables

Apart from prototypes, we are going to develop two UML models to help us towards our goal of understanding the problem as a whole. The first model is the Use Case Model. This will help us to understand what the system needs to do, and what it should look like to the "outside world" (i.e. the users or perhaps the systems it must interface to).

The second model is the Conceptual Model. This model allows us to capture, using UML, a graphical statement of the customer's problem. It will describe all of the major "concepts" in the customer's problem, and how they are related. To build this, we'll use the UML Class Diagram.

Notes We will use this Conceptual Model in the Construction Phase to build our software classes and objects.

Task Compare and contrast feasible time and required time.

Self Assessment

Fill in the blanks:

1. ..... describe how system behaves from the user's point of view.
2. ..... is used to automate manual process.
3. ..... is used to combine functionality from different system
4. ..... is the point where you want to have a better understanding of the problem.
5. A key activity in the Elaboration Phase is the mitigation of .....
6. ..... difficult or problematic areas of the project are a tremendous help in the mitigation of risks.
7. ..... Model allows us to capture, using UML, a graphical statement of the customer's problem.

10.2 Preparing a Problem Statement

Notes

A problem statement is a clear concise description of the issues that need to be addressed by a problem solving team and should be presented to them (or created by them) before they try to solve the problem. When bringing together a team to achieve a particular purpose efficiently provide them with a problem statement. A good problem statement should answer these questions:

1. ***What is the problem?*** This should explain why the team is needed.
2. ***Who has the problem or who is the client/customer?*** This should explain who needs the solution and who will decide the problem has been solved.
3. ***What form can the resolution be?*** What is the scope and limitations (in time, money, resources, and technologies) that can be used to solve the problem? Does the client want a white paper? A web-tool? A new feature for a product? A brainstorming on a topic?

The primary purpose of a problem statement is to focus the attention of the problem solving team. However, if the focus of the problem is too narrow or the scope of the solution too limited the creativity and innovation of the solution can be stifling.

The development of a software system is usually just a part of finding a solution to a larger problem. The larger problem may entail the development of an overall system involving software, hardware, procedures, and organizations. Object oriented programming languages provide a powerful tool for building flexible and extensible software components. However, maximum benefits are gained only if the software is appropriately designed. The choice of classes, and the distribution of tasks between the objects, is of crucial importance.

Object-oriented analysis (OOA) looks at the problem domain, with the aim of producing a conceptual model of the information that exists in the area being analyzed. Analysis models do not consider any implementation constraints that might exist, such as concurrency, distribution, persistence, or how the system is to be built. Implementation constraints are dealt with during object-oriented design (OOD). Analysis is done before the Design.

OOA specifies the structure and the behaviour of the object that comprise the requirements of that specific object. Different types of models are required to specify the requirements of the objects. These object models contain the definition of objects in the system, which includes: the object name, the object attributes, and the objects relationships to other objects. As you know, an object is a representation of a real-life entity, or an abstraction.

For example, objects in a flight reservation system might include: an airplane, an airline flight, an icon on a screen, or even a full screen with which a travel agent interacts.

The sources for the analysis can be a written requirements statement, a formal vision document, and interviews with stakeholders or other interested parties. A system may be divided into multiple domains, representing different business, technological, or other areas of interest, each of which are analyzed separately.

The result of object-oriented analysis is a description of what the system is functionally required to do, in the form of a conceptual model. That will typically be presented as a set of use cases, one or more UML class diagrams, and a number of interaction diagrams. It may also include some kind of user interface mock-up.

Many project teams make the mistake of trying to the analysis and design for all the system usage scenarios before and coding begins. This if often referred to as “big design up front” or BDUF. The problem with BDUF is that in any product design process there is a chance you will get it wrong. Without meaningful testing and feedback, your ultimate design is likely to be very

Notes

flawed. One myth about UML and OOAD is that you need a complete design in order to write code. But that is not right.

Caution You just need enough of the design to get you started.

Did u know? Designing and implementing software one usage scenario at a time, so that users can provide feedback on the end product, is a smart way of fleshing out a better design.

10.2.1 Issues in Complexity Modelling and Abstraction

Divide and Rule has been a fruitful technique of mastering complexity since ancient time. When designing complex software system it is essential to decompose it into smaller and smaller parts each of which we may then redefine independently. Object technologies leads to reuse and reuse leads to faster software development and higher quality software products.

As Brooks suggests, "The complexity of software is an essential property, not an accidental one". We observe that this inherent complexity derives from four elements: the complexity of the problem domain, the difficulty of managing the development process, the flexibility possible through software, and the problems of characterizing the behavior of discrete systems.

The problems we try to solve in software often involve elements of inescapable complexity, in which we find a myriad of competing, perhaps even contradictory, requirements. The external complexity usually springs from the "communication gap" that exists between the users of a system and its developers: Users generally find it very hard to give precise expression to their needs in a form that developers can understand. In some cases, users may have only vague ideas of what they want in a software system. This is not so much the fault of either the users or the developers of a system; rather, it occurs because each group generally lacks expertise in the domain of the other.

A further complication is that the requirements of a software system often change during its development, largely because after seeing early products, such as design documents and prototypes, and then using a system once it is installed and operational are forcing functions that lead users to better understand and articulate their real needs. At the same time, this process helps developers master the problem domain, enabling them to ask better questions that illuminate the dark corners of a system's desired behavior.

The fundamental task of the software development team is to shield users from this vast and often arbitrary external complexity. Software offers the ultimate flexibility, so it is possible for a developer to express almost any kind of abstraction. Engineers try to design the systems with a separation of concerns, so that the behavior in one part of a system has minimal impact on the behavior in another.

Generally, 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. It is important to realize that the architecture of a complex system is a function of its components as well as the hierarchic relationships among these components.

Complex systems have common patterns. These patterns may involve the reuse of small components of larger structures

Notes

Did u know? The discovery of common abstractions and mechanisms greatly facilitates the understanding of complex systems.

Most interesting systems do not embody a single hierarchy; instead, many different hierarchies are usually present within the same complex system.

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

As we work to bring organization to this complexity through the process of design, we must think about many things at once. We are thus faced with a fundamental dilemma.

Notes The complexity of the software systems we are asked to develop is increasing, yet there are basic limits on our ability to cope with this complexity.

When designing a complex software system, it is essential to decompose it into smaller and smaller parts, each of which we may then refine independently. In this manner, we satisfy the very real constraint that exists on the channel capacity of human cognition:

- To understand any given level of a system, we need only comprehend a few parts (rather than all parts) at once.

Self Assessment

Fill in the blanks:

8. A ..... is a clear concise description of the issues that need to be addressed by a problem solving team and should be presented to them before they try to solve the problem.

9. ..... languages provide a powerful tool for building flexible and extensible software components.

10. ..... looks at the problem domain, with the aim of producing a conceptual model of the information that exists in the area being analyzed.

11. The development of a ..... system is usually just a part of finding a solution to a larger problem.

12. Implementation ..... are dealt with during object-oriented design (OOD).

13. A system may be divided into multiple....., representing different business, technological, or other areas of interest, each of which are analyzed separately.

14. The external complexity usually springs from the "....." that exists between the users of a system and its developers.

15. When designing a ..... software system, it is essential to decompose it into smaller and smaller parts, each of which we may then refine independently.

Notes

Case Study

Problem Statement for Railway Reservation System

Software has to be developed for automating the manual railway reservation system. The system should be distributed in nature. It should be designed to provide functionalities as explained below:

1. ***Reserve Seat:*** A passenger should be able to reserve seats in the train. A reservation form is filled by the passenger and given to the clerk, who then checks for the availability of seats for the specified date of journey. If seats are available then the entries are made in the system regarding the train name, train number, date of journey, boarding station, destination, person name, sex and total fare. Passenger is asked to pay the required fare and the tickets are printed. If the seats are not available then the passenger is informed.
2. ***Cancel Reservation:*** A passenger wishing to cancel a reservation is required to fill a form. The passenger then submits the form and the ticket to the clerk. The clerk then deletes the entries in the system and changes the reservation status of that train. The clerk crosses the ticket by hand to mark as cancelled.
3. ***Update Train Information:*** Only the administrator enters any changes related to the train information like change in the train name, train number, train route, etc. in the system.
4. ***Report Generation:*** Provision for generation of different reports should be given in the system. The system should be able to generate reservation chart, monthly train report etc.
5. ***Login:*** For security reasons all the users of the system are given a user id and a password. Only if the id and password are correct is the user allowed entry to the system and select from the options available in the system.
6. ***View Reservation Status:*** All the users should be able to see the reservation status of the train online. The user needs to enter the train number and the pin number printed on his ticket so that the system can display his current reservation status like confirmed, RAC or Wait-listed.
7. ***View Train Schedule:*** Provision should be given to see information related to the train schedules for the entire train network. The user should be able to see the train name, train number, boarding and destination stations, duration of journey etc.

Question

Discuss the case.

Source: http://www.egyankosh.ac.in/bitstream/123456789/16412/3/SECTION%201.pdf

10.3 Summary

- System conception is considered as a requirement analysis phase.
- Requirements describe how system behaves from the user's point of view. True customer requirements should be separated from design decisions.
- Elaboration is the point where you want to have a better understanding of the problem.

Notes

- The purpose of elaboration is to analyse the problem, develop the project plan further, and eliminate the riskier areas of the project.
- A key activity in the Elaboration Phase is the mitigation of risks. The sooner risks are identified and shot down, the lesser their impact will be on the project.
- A problem statement is a clear concise description of the issues that need to be addressed by a problem solving team and should be presented to them (or created by them) before they try to solve the problem.
- Object-oriented analysis (OOA) looks at the problem domain, with the aim of producing a conceptual model of the information that exists in the area being analyzed.
- The result of object-oriented analysis is a description of what the system is functionally required to do, in the form of a conceptual model.

10.4 Keywords

Conceptual model: Conceptual model allows us to capture, using UML, a graphical statement of the customer's problem.

Elaboration: Elaboration is the point where you want to have a better understanding of the problem.

Object oriented analysis: *Object*-oriented analysis (OOA) looks at the problem domain, with the aim of producing a conceptual model of the information that exists in the area being analyzed.

Problem statement: A problem statement is a clear concise description of the issues that need to be addressed by a problem solving team and should be presented to them (or created by them) before they try to solve the problem.

System Conception: System conception is considered as a requirement analysis phase where requirements describe how system behaves from the user's point of view.

10.5 Review Questions

1. Discuss the concept of system conception.
2. Analyze some methods to find new system concepts.
3. What is elaboration? Discuss with example.
4. Analyze the purpose of elaboration.
5. "A key activity in the Elaboration Phase is the mitigation of risks." Comment.
6. Make distinction between use case model and conceptual model.
7. How does prototyping helps in mitigation of risks?
8. Illustrate the concept of preparing a problem statement.
9. Elucidate what is to be done while defining the problem statement.
10. Discuss the issues included in complexity modelling and abstraction.

Answers: Self Assessment

1. Requirements
2. Automation
3. Integration
4. Elaboration

Notes

5. risks
6. Prototyping
7. Conceptual
8. problem statement
9. Object oriented programming
10. Object-oriented analysis (OOA)
11. software
12. Constraints
13. domains
14. communication gap
15. complex

10.6 Further Readings

Books

Booch, Grady, (1994), *Object-oriented Analysis & Design*, Addison Wesley

Laganiere, Robert, (2004), *Object-oriented Software Engineering*, TMH

Rumbaugh, J. (2007),*Object-oriented Modelling and Design with UML*, Pearson Education

Satzinger, (2007), *Object-oriented Analysis & Design with the Unified Process*, Thomson

Online links

http://www.cs.kau.se/~gustas/student/OOAnalysis&Design/Analysis-DesignProcess4.pdf

http://www.enel.ucalgary.ca/People/far/Lectures/SENG401/PDF/OOAD_with_UML.pdf

http://www.newthinktank.com/2013/01/object-oriented-design-tutorial-2/

http://www.scribd.com/doc/55252545/828-Object-Oriented-Modeling-and-Design-3

Unit 11: Domain Analysis

Notes

CONTENTS

Objectives

Introduction

11.1 Domain Class Model

11.1.1 Find Classes

11.1.2 Keeping the Right Classes

11.1.3 Preparing Data Dictionary

11.1.4 Finding Association

11.1.5 Criteria for Keeping Right Association

11.1.6 Finding Attributes

11.1.7 Keeping Right Attributes

11.1.8 Refining with Inheritance

11.2 Domain State Model

11.2.1 Identify Domain Class with State

11.2.2 Find State

11.2.3 Finding Events

11.2.4 Building State Diagram

11.3 Domain Interaction Model

11.4 Iterating the Analysis

11.4.1 Refine the Analysis Model

11.4.2 Restating the Requirements

11.4.3 Analysis & Design

11.5 Summary

11.6 Keywords

11.7 Review Questions

11.8 Further Readings

Objectives

After studying this unit, you will be able to:

- Define Domain Class Model
- Explain domain state model
- Discuss domain interaction model
- Describe the concept of iterating the analysis

Notes

Introduction

Domain model illustrates meaningful conceptual classes in a problem domain. It is a representation of real-world concepts, not software components. It is not a set of diagrams describing software classes, or software objects and their responsibilities. Its development entails identifying a rich set of conceptual classes, and is at the heart of object oriented analysis. It is a visual representation of the decomposition of a domain into individual conceptual classes or objects. It is a visual dictionary of noteworthy abstractions.

OO domain analysis model contain class models, often state models, but seldom has an interaction model. The goal is to analyze a problem without introducing bias for implementation. Business experts should validate the analysis model. Analysis models can be used as an effective means of communication among business experts and system design experts.

11.1 Domain Class Model

Domain class model perform following steps to construct domain class model

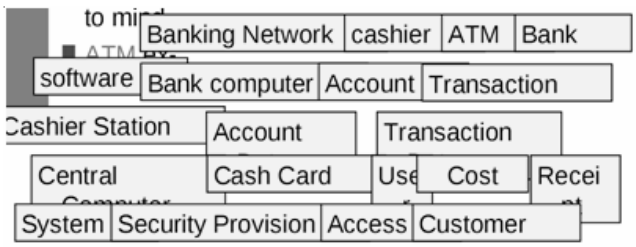
- Find classes
- Prepare data dictionary
- Find association
- Find attribute of objects & links
- Combine classes using inheritance
- Verify that access paths exits
- Iterate & refine model
- Reconsider the level of abstraction
- Group classes into packages

11.1.1 Find Classes

- Find classes for objects from application domain
- All classes must make sense in application sense; avoid computer aspects only consider physical entities.
- Begin by listing classes found in problem statement, write down every classes that comes to mind

Example: Let us see an example of ATM.

Figure 11.1: ATM Example



Source: http://svbitce2010.weebly.com/uploads/8/4/4/5/8445046/ch_11___12_system_conception_domain_analysis.pdf

Notes

- Design a software to support a computerized banking network including both human cashiers and automatic teller machine (ATM) shared by consortium of banks. Each bank provides its own computer to maintain its own account and process transaction against them. Cashier stations are owned by individual banks and communicate directly with their own bank's computer. Human cashier enter account and transaction data
- ATM communicates with central computer. It accepts cash card, interacts with user, communicate with central system to carry out transaction, dispense cash and prints receipt. The system requires appropriate recordkeeping and security provisions. The system must handle concurrent access to same account correctly.
- Bank provides their own s/w and you have to design s/w for ATM. The cost of shared system will be apportioned to banks according to no. of customers with cash card.

11.1.2 Keeping the Right Classes

Redundant Classes

If two classes express the same concept, keep the most descriptive name.

Example: Customer describe person taking an airline flight, passenger is more descriptive. Here Customer and user are redundant. We retain Customer

Irrelevant Classes

- If class has little or nothing to do with problem eliminate it.
- Here Cost is outside the scope of ATM s/w

Vague Classes

- A class should be specific.
- Some tentative class may have ill defined boundaries

Attributes

Names that describe individual objects are attributes.

Example: Account data describes account. ATM dispenses cash & receipts that are peripheral to problem; so treat it as attribute

Operation

If a name describes operation applied to object; then it is not a class

Example: Telephone call is a sequence of actions involving a caller & telephone network so call is a part of state model not a class

Roles

One physical entity corresponds to several classes.

Notes

Example: Person and employee for company database of employee are identical but for govt. tax database, the two are distinct.

Implementation Constructs

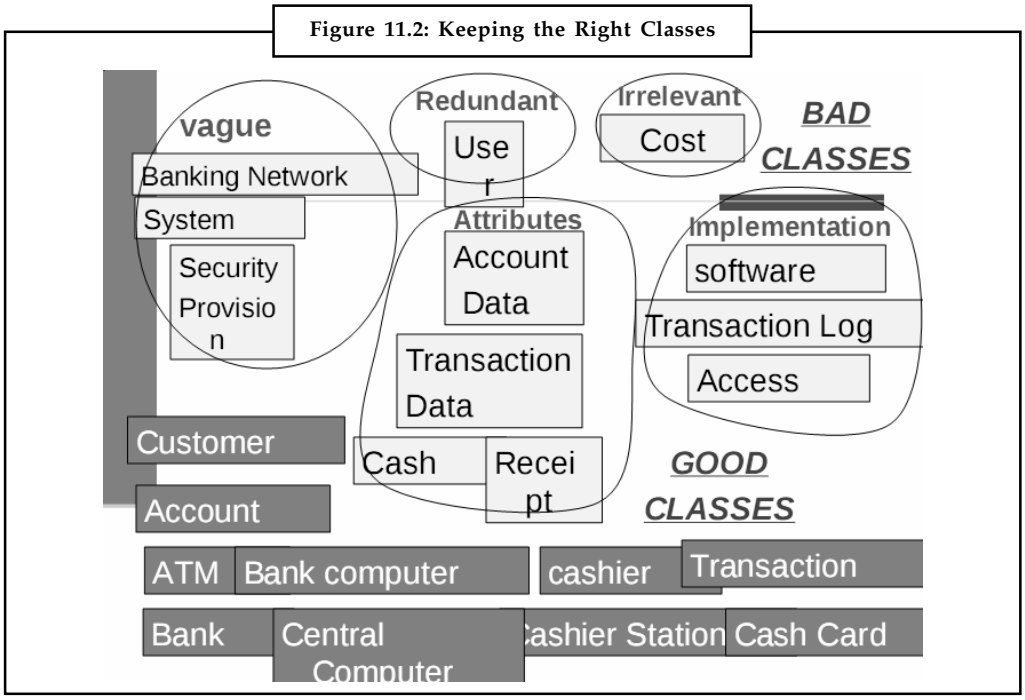
- eliminate extra constructs that are needed later during design but not now
- subroutine, algorithm, process and data structure are implementation constructs

Example: Transaction log is a set of transaction; its exact representation is design issue.

Derived Class

A class that can be derived from other classes is a derived class.

Figure 11.2: Keeping the Right Classes



Source: http://svbitce2010.weebly.com/uploads/8/4/4/5/8445046/ch_11___12_system_ conception _domain_analysis.pdf

11.1.3 Preparing Data Dictionary

Prepare data dictionary for all modeling elements.

- Isolated words have many interpretations; so prepare a data dictionary for all modeling entities.
- Describe the scope of the class within the current problem, including assumptions or restrictions on its membership or use.

Did u know? The data dictionary also describes associations, attributes, and operation.

Notes

11.1.4 Finding Association

- Reference from one class to another

Example: Class person and company relate them with Works for.

- Means verbs or phrases – physical phrase (next to, part of, contained), directed action , communication (talks to), ownerships (has, part of),some condition (works for, manages)

11.1.5 Criteria for Keeping Right Association

Association between Eliminated Classes

If you eliminate one class in association, you must eliminate association

Example: Eliminate Banking network includes cashier station and ATMs

Irrelevant /Implementation Association

Outside the problem domain or deals with implementation constructs

Example: System handles concurrent access is implementation concept.

Actions

Association describe structural property, not an event for object

Example: ATM accepts card describes part of interaction between ATM & customer not a permanent relationship between them.

Ternary Association

Decompose most association among three or more classes into binary association.

Example: Bank computer processes transaction against account can be broken into:

- Bank computer processes transaction
- Transaction concerns account

Derived Association

- Omit association that can be defined in terms of other association because they are redundant

Example: Grandparent of can be defined in terms of pair of Parents of.

- Classes, attributes and associations in class model represent independent information.
- Multiple paths between classes indicates derived association

Notes

Caution Although derived association do not add information, they are useful in real world and in design.

11.1.6 Finding Attributes

- Attributes are data properties of individual objects. Attribute values should not be a class.
- Only consider attributes directly relevant to application. Be sure to give each attribute a meaningful name.
- Omit derived attribute.

11.1.7 Keeping Right Attributes

Objects: If independent existence of element is important than value consider it as object.

Example: Boss refers to a class and salary is an attribute

Qualifier: If value of attribute depends on particular context consider it as qualifier.

Example: Employee no is qualifier

Names: Name is an attribute when it does not depend on context & it is not unique.

Example: Dept. name, person name

Identifiers: Do not include attribute whose only purpose is to identify an object.

Example: Transaction ID is not attribute

Internal Value:

- If attribute describes internal state of object then eliminate it.
- Omit minor attribute that are unlikely to affect most operation

11.1.8 Refining with Inheritance

This step is to organize classes by using inheritance to share common structure

- Inheritance can be discovered by searching classes by similar attributes, associations and operations.
- Organize classes by using inheritance to share common structure
- Inheritance can be added in two directions
- By generalizing classes into super class
- By specializing existing classes into multiple subclasses

Notes

Notes Inheritance can be added by generalizing existing classes into a superclass or by specialising a class into subclasses (based on taxonomic relations).

Bottom-up Generalization

It is done by generalizing common aspect of existing classes into superclasses.

- Discover inheritance from bottom-up searching for classes with similar attributes, association & operation
- Define a super class to share a common feature.

Example: Remote Transaction & CashierTransaction are similar except initiation so they can be generalized by Transaction

Top-down Specialization

It is done by refining existing classes into specialized subclasses.

Look for noun phrases composed of various adjectives on class name

Example: Fixed menu, pop-up menu, sliding menu

Multiple Inheritance

Use it to increase sharing but only if necessary because it increases both conceptual and implementation complexity

Similar Association

When same association name appears more than once with same meaning, generalize it.

Task Compare and contrast bottom-up generalization and top-down specialization.

Self Assessment

Fill in the blanks:

1. Names that describes individual objects are .....
2. A class that can be derived from other class is a .....
3. ..... describe structural property, not an event for object.
4. If independent existence of element is important than value, it is considered as an .....

Notes

5. If value of attribute depends on particular context, it is considered as .....
6. ..... is an attribute when it does not depend on context & it is not unique.
7. ..... can be added by generalizing existing classes into a superclass or by specializing a class into subclasses.
8. ..... is done by generalizing common aspect of existing classes into superclasses.
9. ..... is done by refining existing classes into specialized subclasses.

11.2 Domain State Model

- Objects passes through distinct states during their lifetime.
- Describes various states of objects,
- Properties & constrains of object in various state
- Events that take object from one state to Another

Steps for constructing a domain state model:

- Identify class with state
- Find state
- Find event
- Build state diagram

11.2.1 Identify Domain Class with State

- Check list of domain classes for those that have distinct life cycle.
- Identify significant states in life cycle of object
- Being written Under consideration Accepted or rejected

Example: ATM:-Account class has lifecycle & ATM depends on state of an account

11.2.2 Find State

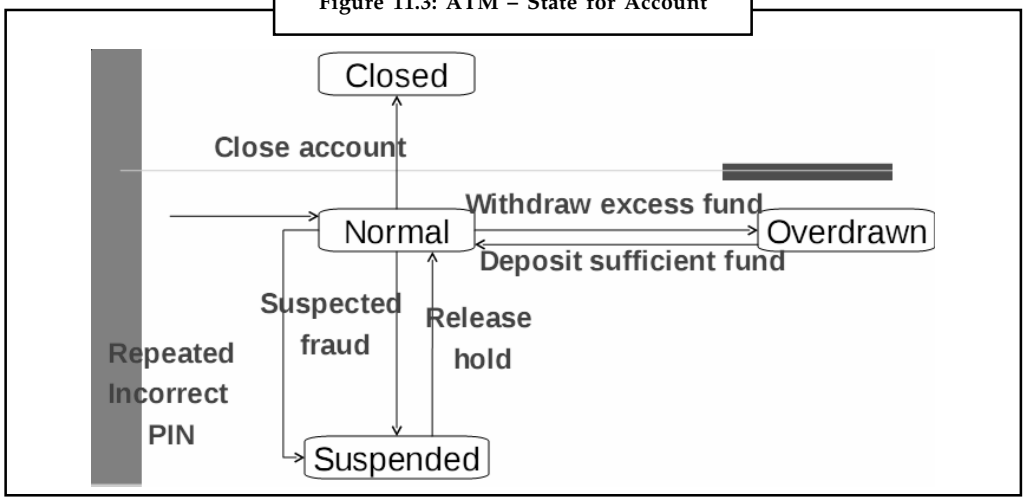
- List states for each class
- Give each state a meaningful name
- State should be based on qualitative difference in behavior, attributes & association

Example: In figure below, we have shown the different states.

- Normal (ready for normal access)
- Closed (closed by customer but still on file in bank records)
- Overdrawn (customer withdrawals exceed the balance in account)
- Suspended (access to account is blocked for some reason)

Notes

Figure 11.3: ATM – State for Account



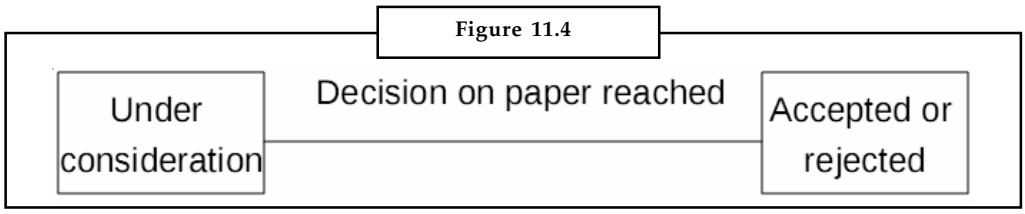
Source:http://svbitce2010.weebly.com/uploads/8/4/4/5/8445046/ch_11___12_system_conception_domain_analysis.pdf

11.2.3 Finding Events

- Find event that causes transition among states.
- Consider those events that occurs within state but do not cause transition
- You can regard event as do-activity

Example: Scientific paper for journal

Figure 11.4



Source: http://svbitce2010.weebly.com/uploads/8/4/4/5/8445046/ch_11___12_system_conception_domain_analysis.pdf

Example: ATM: - close account, repeated incorrect PIN, Administrative action

11.2.4 Building State Diagram

- Note state to which event applies.
- Add transitions to show change in state caused by event.

Caution If a event has different effect in different state add a transition for each state

Task Make distinction between closed state and suspended state.

Notes

Self Assessment

Fill in the blanks:

10. In ..... model, objects passes through distinct states during their lifetime.
11. ..... are added to show change in state caused by event.

11.3 Domain Interaction Model

The interaction model is not very significant for domain analysis. During domain analysis, the stress is on key concepts and deep structural relationships and not the user's view of them.

Notes The interaction model is a significant aspect of application modelling.

Building interaction model include the following steps:

- Determine system boundary.
- Identify actors and use cases.
- Determine initial and final events in each use case.
- Define scenarios for normal course of events.
- Define alternative scenarios.
- Identify external events.
- Prepare activity diagrams for use cases.
- Identify dependencies among actors and use cases.
- Consistency checking against the domain model.

Self Assessment

Fill in the blanks:

12. During ....., the stress is on key concepts and deep structural relationships and not the user's view of them.
13. The first step in interaction model is to determine .....

11.4 Iterating the Analysis

Iterating the analysis requires more than one pass to complete. Problem statement often contains circularities and cannot be approached in a linear way because different parts of problem interact. Prepare a model first and then iterate it as your understanding increases.

Did u know? Final analysis is verified with requestor and application domain expert.

11.4.1 Refine the Analysis Model

- Overall analysis model shows inconsistencies and imbalance.

Notes

- Try to refine classes to increase sharing and improve structure.
- Some constructs won't fit in model; You probably missed or miscast a general concept; reexamine them & change model
- Include exception, special case
- Remove classes or association that seemed to be useful at first but now appears extra; you can combine two classes in analysis can be combined
- A good model have few small areas & don't have extra details

11.4.2 Restating the Requirements

- Most of real requirements will be part of model

  Other requirements specify method of solution and should be separated if possible.
- During analysis some requirements may appear to be incorrect & impractical; confirm correction to requirements
- Business expert verify it to make sure that it correctly models the real world
- Final model serves as basis for system architecture, design & implementation.

11.4.3 Analysis & Design

Goal of analysis is to specify the problem without implementation details but it is impossible to avoid all taints of implementation. There is no absolute line between various development stages nor is there any such thing as a perfect analysis.

Self Assessment

Fill in the blanks:

14. ..... the analysis requires more than one pass to complete.
15. Final analysis is verified with requestor & ..... expert.

11.5 Summary

- The goal of the domain analysis model is to analyze a problem without introducing bias for implementation.
- All classes must make sense in application sense; avoid computer aspects only consider physical entities.
- If two classes express same concept keep the most descriptive name. Names that describe individual objects are attributes.
- Prepare data dictionary for all modeling elements and describe scope of class, restriction on its use, associations, attributes, operations.
- If independent existence of element is important than its value, consider it as object.
- Name is an attribute when it does not depend on context & it is not unique. Inheritance can be added by generalizing existing classes into a superclass or by specializing a class into subclasses (based on taxonomic relations).

Notes

- Use multiple inheritance to increase sharing but only if necessary because it increases both conceptual and implementation complexity. In domain state model, Objects passes through distinct states during their lifetime.
- During domain analysis, the stress is on key concepts and deep structural relationships and not the user's view of them. However, the interaction model is a significant aspect of application modeling. Iterating the analysis requires more than one pass to complete.

11.6 Keywords

Analysis: It specifies the problem without implementation details.

Attributes: Attributes are data properties of individual objects.

Derived class: A derived class is a class that can be derived from other class.

Domain analysis model: It analyzes a problem without introducing bias for implementation.

Irrelevant classes: Irrelevant classes are those which have little or nothing to do with problem.

Name: Name is an attribute when it does not depend on context.

Object: If independent existence of element is important than value, then it is considered as an object.

Qualifier: If value of attribute depends on particular context, it is considered as a qualifier.

11.7 Review Questions

1. Describe the concept of domain analysis model.
2. Analyze the steps taken in order to construct domain class model.
3. How do you find classes to construct domain class model? Illustrate with example.
4. Discuss the concept of keeping the right classes. Give example.
5. Make distinction between ternary Association and derived association.
6. Illustrate the steps used for finding attributes.
7. Illustrate the steps for bottom-up generalization and top-down specialization.
8. What is a domain state model? Explain.
9. How do you find state for constructing a domain state model? Illustrate.
10. Analyze the steps included in building interaction model.

Answers: Self Assessment

1.	Attributes	2.	derived class
3.	Association	4.	Object
5.	qualifier	6.	Name
7.	Inheritance	8.	Bottom-Up Generalization

Notes

9. Top-Down Specialization
10. domain state
11. Transitions
12. domain analysis
13. system boundary
14. Iterating
15. application domain

11.8 Further Readings

Books

Booch, Grady, (1994), *Object-oriented Analysis & Design,* Addison Wesley

Laganiere, Robert, (2004), *Object-oriented Software Engineering,* TMH

Rumbaugh, J. (2007), *Object-oriented Modelling and Design with UML,* Pearson Education

Satzinger, (2007), *Object-oriented Analysis & Design with the Unified Process,* Thomson

Online links

http://g.oswego.edu/dl/oosdw3/ch13.html

http://www.cs.kau.se/~gustas/student/OOAnalysis&Design/Analysis-DesignProcess4.pdf

http://www.exforsys.com/tutorials/ooad/ooad-analysis.html

https://blog.itu.dk/BDSA-E2012/files/2012/08/bdsa2.pdf

Notes

Unit 12: System Design

CONTENTS

Objectives

Introduction

12.1 System Design Overview

12.2 System Design Decisions

12.2.1 Reuse Plan

12.2.2 Breaking a System into Subsystems

12.2.3 Identifying Concurrency

12.2.4 Allocation

12.2.5 Management of Data Storage

12.2.6 Handling Global Resources

12.2.7 Software Control Strategy

12.2.8 Boundary Conditions

12.2.9 Setting Trade-off Priorities

12.2.10 Common Architectural Styles

12.3 Summary

12.4 Keywords

12.5 Review Questions

12.6 Further Readings

Objectives

After studying this unit, you will be able to:

- Describe overview of system design
- Discuss the concept of reuse plan
- Define concurrency and allocation
- Explain the concept of software control strategy
- Discuss handling boundary conditions

Introduction

After analyzing the problem, one must decide how to approach the system design. During system design developers devise the high-level strategy (system architecture) for solving the problem and building a solution and make decisions about the organization of the system into subsystems, the allocation of subsystems to hardware and software and major policy decisions that form the basis for class design. System design is the first design state for devising the basic approach to solving the problem. During system design, developers decide the overall structure

and style. In this unit, we will discuss the concept of system design. Also, we will discuss various design decisions that are to be made during system design such as reuse plan, identifying concurrency, allocation, software control strategy, etc.

12.1 System Design Overview

There are different approaches of System design Overview concerning various opinion. Let us see some of them.

1. During system design the high-level of system architecture is chosen, the system is divided into sub systems and concurrency is identified. Two objects are inherently concurrent if they receive events at the same time and do not interact. If the events not are synchronized the two objects cannot be on a single thread of control.

   A thread of control is path through a set of state diagrams. Only a single object at a time may be active on a thread. Sub systems are allocated to processors and tasks. Estimate hardware resource requirements to decide if some subsystem shod be implemented in hardware rather than in software. Allocate tasks to processor and determine physical connectivity. Choose a strategy for management of data stores. Data stores are used to give data some degree of permanency and to clearly separate subsystems. Data stores may be implemented by using files or by databases. Identify and handle access to global resources. Choose an approach to implement software control. Consider boundary conditions including the issues of initialization, termination, and failure. See if a common architectural framework might fit the application. Some common architectural frame works are:

   (a) Batch transformation – a data transformation is executed once on an entire input set.

   (b) Continuous transformations – a data transformation performed continuously as inputs change.

   (c) Interactive interfaces – a system dominated by external interactions.

   (d) Dynamic simulations – a system that simulates evolving real-world objects.

   (e) Transaction managers – a system concerned with storing and updating data, often including concurrent access from different physical locations.

2. During system analysis, the focus is on what needs to be done, independent of how it is done. During design, developers make decisions about how the problem will be solved, first a high level and then with more detail.

   The system architecture determines the organization of the system into subsystems.

Notes A rough performance estimate ("Back of the envelop") should be calculated for a new system. The purpose is not to achieve high accuracy, but merely to determine if the system is feasible. The calculation should be fast and should involve common sense.

Self Assessment

Fill in the blanks:

1. During system design the high-level of ..... is chosen.
2. A thread of control is path through a set of .....

Notes

3. ..... is the first design state for devising the basic approach to solving the problem.
4. The system architecture determines the organization of the system into .....

12.2 System Design Decisions

During system design, the design decisions that are to be made are discussed in the unit.

12.2.1 Reuse Plan

Reuse is often cited as an advantage of OO technology. But reuse does not happen automatically. There are two different aspects – using existing things and creating reusable new things. It is much easier to use existing things than to design new things for uncertain use to come. Reusable things include models, libraries, frameworks and patterns. Reuse of models is often the post practical form of reuse. The logic in a model can apply to multiple problems.

A Library is a collection of classes that are useful in many contexts.

Caution The collection of classes must be carefully organized, so that users can find them.

There must be several qualities for a good library:

1. Coherence
2. Completeness
3. Consistency
4. Efficiency
5. Extensibility
6. Generality
7. Problems related with libraries
8. Argument validation
9. Error handling
10. Control paradigms
11. Group operations
12. Garbage collection
13. Name collisions

Frameworks based Reuse

A frame work is skeletal structure of a program that must be elaborated to build a complete application. This elaboration often consists of specializing abstract classes with behavior specific to an individual application. A class library may accompany a frame work.

Pattern based Reuse

A pattern is a proven solution to general problem. Various patterns target different phases of the software development life cycle. There are patterns for analysis, architecture, design and implementation. A pattern is different from framework. A pattern is typically a small number

Notes

of classes and relationships where as a framework is much broader in scope and covers an entire subsystem or application.

12.2.2 Breaking a System into Subsystems

Each major piece of the system is called subsystem, which depends on some there. In a subsystem classes share common properties, have similar functionality, have the same physical location, or execute on the same hardware. A subsystem is a package of classes, associations, operations, events and constraints that are interrelated and have a reasonably well defined interface to the rest of the system. The interface specifies all interactions with the subsystem to allow independent subsystem design.

Did u know? A subsystem is usually identified by the services, which is a group of related functions that share some common purpose.

The decomposition of systems into subsystems may be organized as a sequence of horizontal layers of vertical partitions.

Layers

Layers define an abstract world and work like a client of services for layers below and as a supplier of services for layers above it. A layered system is an ordered set of virtual worlds, each built in terms of the ones below it and providing the implementation basis for the ones above it. Layered architecture comes in closed or open.

In a closed architecture, each layer is built only in terms of the immediate lower layer. This reduces dependencies between layers and allows changes to be made most easily. In opened architecture a layer knows of all layers below. It means a layer can use the features of any lower layer to any depth. This reduces the need to redefine operations at each level.

Figure 12.1: Layered Architecture

Interactive Graphics Application
Windows Operations
Screen Operations
Pixel Operations
Device I/O Operations

Closed Architectures

- Each layer is built only in terms of the immediate lower layer
- Reduces dependencies between layers
- Facilitates change

Open Architectures

- Layer can use any lower layer
- Reduces the need to redefine operations at each level
- More efficient/compact code
- System is less robust/harder to change

Notes

Properties of Layered Architectures

- Top and bottom layers specified by the problem statement
  - Top layer is the desired system
  - Bottom layer is defined by available resources (e.g. HW, OS, libraries)
- Easier to port to other HW/SW platforms

Task Make distinction between closed architecture and open architecture.

Partitions

Partitions vertically divide a system into several independent or weakly subsystems, each providing one kind of service. One difference between layers and partitions is that layers vary in their level of abstraction, but partitions merely divide a system into pieces, all of which have a similar level of abstraction.

You can decompose a system into subsystems by combining layers and partitions. Layers can be partitioned and partitions can be layered. Most large systems require a mixture of layers and partitions.

Partitioned Architectures divide system into weakly-coupled subsystems.

- Each provides specific services
- Vertical decomposition of problem

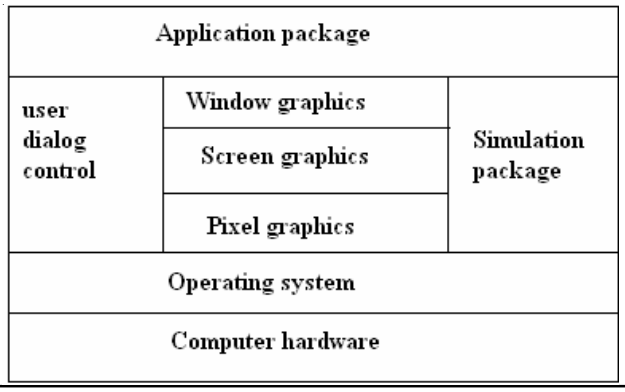
Example: Partitioned Architecture

Figure 12.2: Partitioned Architecture

File System	Process Control	Virtual Memory Management	Device Control

Typical Application Architecture

Figure 12.3: Application Architecture



Notes

12.2.3 Identifying Concurrency

In the analysis model, as the real world and in hardware, all objects are concurrent. In an implementation, not all software objects are concurrent, because one processor may support many objects. One important goal of the system design is to identify the objects that must be active concurrently and the objects that have mutually exclusive activity.

Identifying inherent concurrency: The state model is the guide to identifying concurrency. Two objects are inherently concurrent if they can receive events at the same time without interacting. If events are unsynchronized, you cannot fold the objects onto a single thread of control.

Inherent Concurrency

- May involve synchronization.
- Multiple objects receive events at the same time without interacting.

Example: User may issue commands through control panel at same time that the sensor is sending status information to the Safe Home system.

Determining Concurrent Tasks

- Thread of control
  - Path through state diagram with only one active object at any time
- Threads of control are implemented as tasks
  - Interdependent objects
  - Examine state diagram to identify objects that can be implemented in a task

Concurrency Testing Tools

There are a number of concurrency testing tools on the market to help you deal with potential deadlocks, live locks, hangs, and all the other issues you experience when running parallel transactions. Each tool we look at here will help in a particular area.

1. ***CHESS:*** Created by Microsoft Research, CHESS is a novel combination of model checking and dynamic analysis. It detects concurrency errors by systematically exploring thread schedules and interleaving. It is capable of finding race conditions, deadlocks, hangs, live locks, and data corruption issues. To help with debugging, it also provides a fully repeatable execution. Like most model checking, the systematic exploration provides thorough coverage.

   As a dynamic analysis tool, CHESS runs a regular unit test repeatedly on a specialized scheduler. On every repetition, it chooses a different scheduling order. As a model checker, it controls the specialized scheduler that is capable of creating specific thread interleaving. To control the state space explosion, CHESS applies partial-order reduction and a novel iteration context bounding.

   In iteration context bounding, instead of limiting the state space explosion by depth, CHESS limits number of thread switches in a given execution. The thread itself can run any number of steps between thread switches, leaving the execution depth unbounded (a big win over traditional model checking). This is based on the empirical evidence that a small number of systematic thread switches is sufficient to expose most concurrency bugs.

Notes

CHESS can detect deadlocks and races but relies on programmer assertions for other state verification. It also expects all programs to terminate and that there is a fairness guarantee (forward progress) for all threads. Thus, if the program enters a state of continuous loop, it reports a livelock.

2. ***The Intel Thread Checker:*** This is a dynamic analysis tool for finding deadlocks (including potential deadlocks), stalls, data races, and incorrect uses of the native Windows synchronization APIs. The Thread Checker needs to instrument either the source code or the compiled binary to make every memory reference and every standard Win32 synchronization primitive observable. At run time, the instrumented binary provides sufficient information for the analyzer to construct a partial-order of execution. The tool then performs a "happens-before" analysis on the partial order. Please refer to the "Race Detection Algorithms" sidebar for more information on happens-before analysis.

   For performance and scalability reasons, instead of remembering all accesses to a shared variable, the tool only remembers recent accesses. This helps to improve the tool's efficiency while analyzing long-running applications. However, the side effect is that it will miss some bugs. It's a trade-off, and perhaps it's more important to find many bugs in long-running applications than to find all bugs in a very short-lived application.

   The only other big drawback of the tool is that it can't account for synchronization via interlocked operations, such as those used in custom spin locks.

   Did u know? For applications that use only standard synchronization primitives, this is probably one of the best-supported test tools available for concurrency testing native applications.

3. ***Chord:*** This is a flow-insensitive, context-sensitive static analysis tool for Java. Being flow-insensitive allows it be far more scalable than other static tools, but at the cost of losing precision. It also takes into account the specific synchronization primitives available in Java. The algorithm used is very involved and would require the introduction of many concepts.

4. ***KISS:*** Developed by Microsoft Research, this model checker tool is for concurrent C programs. Since state space explodes quickly in a concurrent system, KISS transforms a concurrent C program into a sequential program that simulates the execution of interleaving. A sequential model checker is then used to perform the analysis.

   The application is instrumented with statements that convert the concurrent program to a sequential program, with KISS assuming the responsibility of controlling the non-determinism. The non-determinism context switching is bounded by similar principles described in CHESS above. The programmer is supposed to introduce asserts which validate concurrency assumptions. The tool does not report false positives. The tool is a research prototype and has been used by the Windows driver team, which primarily uses C code.

5. ***Zing:*** This tool is a pure model checker meant for design verification of concurrent programs. Zing has its own custom language that is used to describe complex states and transition, and it is fully capable of modeling concurrent state machines. Like other model checkers, Zing provides a comprehensive way to verify designs; it also helps build confidence in the quality of the design since you can verify assumptions and formally prove the presence or absence of certain conditions. It also contends with concurrent state space explosion by innovative reduction techniques.

Notes

The model that Zing uses (to check for program correctness) has to be created either by hand or by translators. While some specific domain translators can be written, we have yet to come across any complete and successful translators for native or CLR applications. Without having translators, we believe Zing cannot be used in large software projects, except for verifying the correctness of critical subsections of the project.

12.2.4 Allocation

You must allocate each concurrent subsystem to a hardware unit, either a general-purpose processor or specialized functional unit. The system designer must do the following:

- Estimate performance needs and the resources needed to satisfy them.
- Choose hardware or software implementation for subsystems
- Allocate software subsystems to processors to satisfy performance needs and minimize inter processor communication
- Determine the connectivity of the physical units that implement the subsystems.
- Consider the connection between nodes and communication protocols to be used.
- Consider the need for redundant processing.
- Identify any interface implied by deployment.

UML deployment diagram can be used to present the above-mentioned steps. A deployment diagram shows how the systems will be physically distributed on the hardware.

12.2.5 Management of Data Storage

There are several alternatives for data storage that you can use separately or in combination: data structures, files and databases. Different kinds of data stores provides trade-offs among cost, access time, capacity and reliability.

Database

These are the advantage and disadvantage using database.

Advantages

- Efficient management
- Multi-user support.
- Roll-back support

Disadvantages

- Performance overhead
- Awkward (or more complex) programming interface
- Hard to fix corruption

Flat Files

These are the advantage and disadvantage using File.

Notes

Advantages

- Easy and efficient to construct and use
- More readily repairable

Disadvantages

- No rollback
- No direct complex structure support
- Complex structure requires a grammar for file Format

Flat File Storage and Retrieval

- Useful to define two components (or classes)
  - Reader reads file and instantiates internal object Structure
  - Writer traverses internal data structure and writes out Presentation
- Both can (should) use formal grammar
  - Tools support: Yacc, Lex.

Java Data Marshalling

- Provides a means of "serializing" a set of objects
- Requires classes to implement the "Serializable" Interface.
- Stream can be written/read to a file
- Stream can be written/read to a network socket

Serialization Example:

Figure 12.4: Serialization Example

```
private void
writeObject(java.io.ObjectOutputStream out)
throws IOException;

private void
readObject(java.io.ObjectInputStream in)

throws IOException, ClassNotFoundException;
```

```
FileOutputStream ostream = new
FileOutputStream("t.tmp");

ObjectOutputStream p = new
ObjectOutputStream(ostream);

p.writeInt(12345);

p.writeObject("Today");

p.writeObject(new Date());

p.flush(); ostream.close();
```

Notes

12.2.6 Handling Global Resources

The system designer must identify global resources and determine mechanisms for controlling access to them. There are several kinds of global resources: Physical system, Space, Logical names, Access to shared data.

Identify global resources and determine access patterns.

Example:

- Physical units (processors, tape drives)
- Available space (disk, screen, buttons)
- Logical names (object IDs, filenames)
- Access to shared data (database, file)

12.2.7 Software Control Strategy

It is best to choose a single control style for the whole system. There are two kinds of control flows in a software system: External control and internal control. External control concerns the flow of externally visible events among the objects in the system. There are three kinds of external events: procedural-driven sequential, event-driven sequential and concurrent.

In a procedural-driven sequential system, control within program code. Procedures request external input and then wait for it, when input arrives, control resumes within the procedure that made the call.

The major advantage of procedure-driven control is that it is easy to implement with conventional languages.

Caution The procedure-driven control requires the concurrency inherent in objects to be mapped into a sequential flow of control.

In an event-driven sequential model, control resides within a dispatcher or monitor that the language, subsystem or operating system provides. Developers attach application procedures to events and the dispatcher calls the procedures when the corresponding events occur. It's more flexible.

Notes In a concurrent system, control resides concurrently in several independent objects, each a separate task.

Internal control refers to the flow of control within a process. It exists only in the implementation and therefore is neither inherently concurrent nor sequential.

12.2.8 Boundary Conditions

Although most of the system design concerns steady-state behavior system designer must consider boundary conditions as well and address issues like initialization, termination and failure (the unplanned termination of the system).

Notes

- ***Initialization:*** It refers to initialization of constant data, parameters, global variables, tasks, guardian objects, and classes as per their hierarchy. Initialization of a system containing concurrent tasks must be done in a manner so that tasks can be started without prolonged delays. There is quite possibility that one object has been initialized at an early stage and the other object on which it is dependent is not initialized even after considerable time. This may lead to halting of system tasks.
- ***Termination:*** Termination requires that objects must release the reserved resources. In case of concurrent system, a task must intimate other tasks about its termination.
- ***Failure:*** Failure is the unplanned termination of the system, which can occur due to system fault or due to user errors or due to exhaustion of system resources, or from external breakdown or bugs from external system. The good design must not affect remaining environment in case of any failure and must provide mechanism for recording details of system activities and error logs.

12.2.9 Setting Trade-off Priorities

The system designer must set priorities that will be used to guide trade-offs for the rest of the design.

For example system can be made faster using extra memory.

Design trade-offs involve not only the software itself but also the process of developing it. System designer must determine the relative importance of the various criteria as a guide to making design trade-offs. Design trade-offs affect entire character of the system. Setting trade-offs priorities is at best vague. We cannot even give a full list of design criteria that might be subject to trade-offs.

Summary of the setting trade-off-priorities

- Establish priorities for choosing between incompatible goals
- Implement minimal functionality initially and embellish as appropriate
- Isolate decision points for later evaluation
- Trade efficiency for simplicity, reliability.

12.2.10 Common Architectural Styles

Several prototypical architectural styles are common in existing systems. Each of these is well suited to a certain kind of system. Some of the styles are as follows:

Batch Transformation

A data transformation executed once on an entire input set. It performs sequential computations. The application receives the inputs and the goal is to compute an answer; there is no ongoing interaction with the outside world. The steps are as follows:

1. Break the overall transformation into stages, with each stage performing one part of the transformation.
2. Prepare class models for the input, output and between each pair of successive stages. Each state knows only about the models on either side of it.
3. Expand each stage in turn until the operations are straight forward to implement.
4. Restructure the final pipeline for optimization.

Notes

Continuous Transformation

A data transformation performed continuously as inputs change. It is a system in which the outputs actively depend on changing inputs. It updates outputs frequently. One way to implement continuous transformation is with a pipeline of functions. The steps in designing a pipeline for a continuous transformation are as follows:

- Break the overall transformation into stages performing one part of the transformation.
- Define input, output and intermediate models between each pair of successive stages, as for the batch transformation.
- Differentiate each operation to obtain incremental changes to each stage. That is, propagate the incremental effects of each change to an input through the pipeline as a series of incremental updates.
- Add additional intermediate objects for optimization.

Interactive Interface

A system dominated by external interactions. It is a system dominated by interactions between the system and external agents, such as human or devices. The external agents are independent of the system. The steps in designing an interactive interface are as follows:

- Isolate interface classes from the application classes.
- Use predefined classes to interact with external agents, if possible.
- Use the state model as the structure of the program.
- Isolate physical events from logical events. Often a logical event corresponds to multiple physical events.
- Fully specify the application functions that are invoked by the interface

Dynamic Simulation

It is a system that simulates evolving real-world objects. The steps in designing a dynamic simulation are as follows:

- Identify active real word objects from the class model. These objects have attributes that are periodically updated.
- Identify discrete events, which correspond to discrete interactions with the object.
- Identify continuous dependencies. Real world attributes may be dependent on other real-world attribute.
- Generally a simulation driven by a timing loop at a fine time scale. Discrete events between objects can often be exchanged as part of the timing loop.

Real Time System

A system dominated by strict timing constrains. It is an interactive system with tight time constraints on actions. Transaction Manager is a system concerned with storing and updating data, often including concurrent access from different physical locations. Most transaction manages deal with multiple users who read and write data at the same time.

Notes

Steps are:

- Map the class model to database structures.
- Determine the units of concurrency.
- Determine the unit of transaction.
- Design concurrency control for transactions.

Task Discuss the steps used for batch transformation and continuous transformation.

Self Assessment

Fill in the blanks:

5. A ..... is a collection of classes that are useful in many contexts.
6. A ..... is skeletal structure of a program that must be elaborated to build a complete application.
7. A ..... is a proven solution to general problem.
8. A subsystem is a ..... of classes, associations, operations, events and constraints that are interrelated.
9. ..... define an abstract world and work like a client of services for layers below and as a supplier of services for layers above it.
10. Two objects are inherently ..... if they can receive events at the same time without interacting.
11. ..... is a flow-insensitive, context-sensitive static analysis tool for Java.
12. A ..... diagram shows how the systems will be physically distributed on the hardware.
13. ..... control concerns the flow of externally visible events among the objects in the system.
14. ..... requires that objects must release the reserved resources.
15. ..... is a system that simulates evolving real-world objects.

12.3 Summary

- During system design developers devise the high-level strategy (system architecture) for solving the problem and building a solution and make decisions.
- Reuse is often cited as an advantage of OO technology. Reusable things include models, libraries, frameworks and patterns. Reuse of models is often the post practical form of reuse.
- In a subsystem classes share common properties, have similar functionality, have the same physical location, or execute on the same hardware.

Notes

- Two objects are inherently concurrent if they can receive events at the same time without interacting. If events are unsynchronized, you cannot fold the objects onto a single thread of control.
- You must allocate each concurrent subsystem to a hardware unit, either a general-purpose processor or specialized functional unit.
- The system designer must identify global resources and determine mechanisms for controlling access to them. There are several kinds of global resources: Physical system, Space, Logical names, Access to shared data.
- It is best to choose a single control style for the whole system. There are two kinds of control flows in a software system: External control and internal control.
- Although most of the system design concerns steady-state behavior system designer must consider boundary conditions as well and address issues like initialization, termination and failure.
- The system designer must set priorities that will be used to guide trade-offs for the rest of the design. For example system can be made faster using extra memory.

12.4 Keywords

Deployment diagram: A deployment diagram shows how the systems will be physically distributed on the hardware.

External control: External control concerns the flow of externally visible events among the objects in the system.

Initialization: It refers to initialization of constant data, parameters, global variables, tasks, guardian objects, and classes as per their hierarchy.

Internal control: Internal control refers to the flow of control within a process.

Layers: Layers define an abstract world and work like a client of services for layers below and as a supplier of services for layers above it.

Partitions: Partitions vertically divide a system into several independent or weakly subsystems, each providing one kind of service.

Pattern: A pattern is a proven solution to general problem.

Subsystem: A subsystem is a package of classes, associations, operations, events and constraints that are interrelated and have a reasonably well defined interface to the rest of the system.

12.5 Review Questions

1. Describe the concept of system design.
2. Make distinction between frameworks based reuse and pattern based reuse.
3. Explain how to organize the decomposition of systems into subsystems.
4. Discuss the concept of identifying concurrency with example.
5. What should system designer do in order to provide allocation to each component? Discuss.
6. Analyze and discuss the use of several kinds of global resources.
7. Illustrate the concept of choosing software control strategy.

Notes

8. Explain the boundary conditions that should be considered in system design.
9. "The system designer must set priorities that will be used to guide trade-offs for the rest of the design." Comment.
10. Identify and discuss various tools used for concurrency testing.

Answers: Self Assessment

1.	system architecture	2.	state diagrams
3.	System design	4.	subsystems
5.	Library	6.	framework
7.	pattern	8.	Package
9.	Layers	10.	Concurrent
11.	Chord	12.	Deployment
13.	External	14.	Termination
15.	Dynamic Simulation		

12.6 Further Readings

Books

Booch, Grady, (1994), *Object-oriented Analysis & Design,* Addison Wesley

Laganiere, Robert, (2004), *Object-oriented Software Engineering,* TMH

Rumbaugh, J., (2007), *Object-oriented Modelling and Design with UML,* Pearson Education

Satzinger, (2007), *Object-oriented Analysis & Design with the Unified Process,* Thomson

Online links

http://www.slideshare.net/anandgrewal1/system-design-15438760

http://www.imse.hku.hk/imse1013/pdf/ESA-06%20OOAD.pdf

http://www.iai.uni-bonn.de/III/lehre/vorlesungen/SWT/OOSC06/slides/09%20-%20OOSC%20-%20System%20Design%202-2.pdf

http://ce.sharif.edu/courses/85-86/2/ce924/resources/root/Presentations/2.%20OOAD.pdf

Notes

Unit 13: Class Design

CONTENTS

Objectives

Introduction

13.1 Design Axioms

13.1.1 Corollaries

13.2 Concept of Class Design

13.2.1 Object-oriented Design Philosophy

13.2.2 Class Visibility: Designing well-defined Public, Private and Protected Protocols

13.2.3 Private and Protected Protocol Layers: Internal

13.2.4 Public Protocol Layer: External

13.2.5 Designing Classes: Refining Attributes

13.3 Summary

13.4 Keywords

13.5 Review Questions

13.6 Further Readings

Objectives

After studying this unit, you will be able to:

- Describe the axioms of OOD
- Explain the concept of class design
- Discuss the types of attribute

Introduction

The class design phase determines the full definitions of the classes and associations used in the implementation, as well as the interfaces and algorithms of the methods used to implement operations. The object-designer works to implement the objects discovered during analysis phase. All the operations identified during analysis are expressed as algorithms, with complex operations expressed as internal operations. You need to apply axioms to the design classes, their attributes, structures, associations, protocols and methods. This step includes refinement and completion of static UML class diagram done by adding details to UML class diagram which includes performing the activities:

- Attributes should be refined
- Protocols and methods should be designed by using UML activity diagram to represent the method algorithm

Notes

- Refinement of association between classes (if necessary)
- Refinement of class hierarchy and design with inheritance (if necessary)

It also includes iteration and refinement work to be done again.

13.1 Design Axioms

An axiom is a fundamental truth that always is observed to be valid and for which there is no counter example or exception. They cannot be proven or derived but they can be invalidated by counter examples or exceptions.

A theorem is a proposition that may not be self-evident but can be proved from accepted axioms. A corollary is a proposition that follows from an axiom or another proposition that has been proven.

Axioms of OOD are as follows:

The axiom 1 of object-oriented design deals with relationships between system components (such as classes, requirements and software components) and axiom 2 deals with the complexity of design.

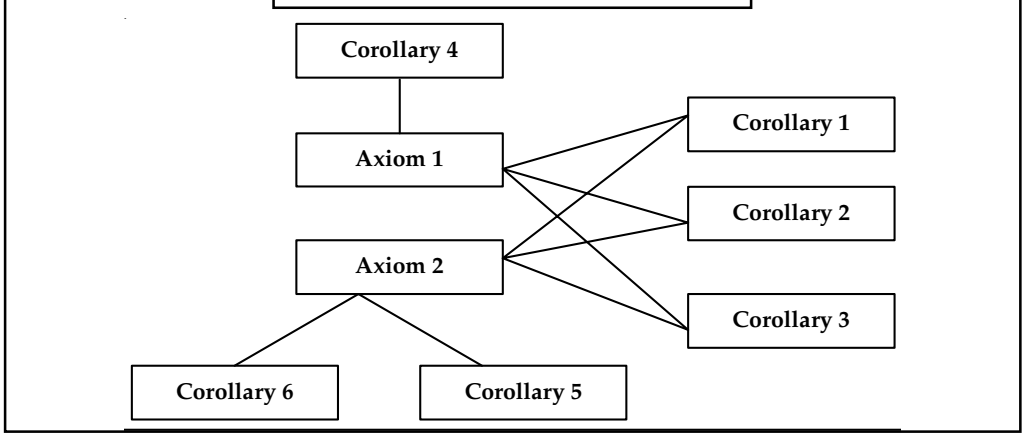
- ***Axiom 1:*** The independence axiom. Maintain the independence of components. According to axiom 1, each component must satisfy its requirements without affecting other requirements. For example, let us design a refrigerator door which can provide access to food and the energy lost should be minimised when the door is opened and closed. Opening the door should be independent of losing energy.
- ***Axiom 2:*** The information axiom. Minimise the information content of the design. It is concerned with simplicity. In object-oriented system, to minimise complexity use inheritance and the system's built in classes and add as little as possible to what already is there.

13.1.1 Corollaries

Corollaries may be called Design rules, and all are derived from the two basic axioms.

The origin of corollaries is shown in Figure 13.1. Corollaries 1, 2 and 3 are from both axioms, whereas corollary 4 is from axiom 1 and corollaries 5 and 6 are from axiom 2.

Figure 13.1: Origin of Corollaries



Source: Bahrami, Ali. *"Object Oriented Systems Development,"* Tata McGraw-Hill Education, 2004

Notes

- ***Corollary 1 – Uncoupled design with less information content:*** Highly cohesive objects can improve coupling because only a minimal amount of essential information need be passed between objects.
- ***Corollary 2 – Single purpose:*** Each class must have a single, clearly defined purpose. While documenting, one should be able to describe the purpose of a class in few sentences.
- ***Corollary 3 – Large number of simple classes:*** Keeping the classes simple allows reusability.
- ***Corollary 4 – Strong mapping:*** There must be a strong association between the physical system (analysis's objects) and logical design (design's object).
- ***Corollary 5 – Standardisation:*** Promote standardization by designing inter changeable components and reusing existing classes or components.
- ***Corollary 6 – Design with inheritance:*** Common behavior (methods) must be moved to super classes. The superclass-subclass structure must make logical sense.

Corollary 1: Uncoupled Design with Less Information Content

Coupling is a measure of the strength of association established by a connection from one object or software component to another. Coupling is a binary relationship. It is important for design because a change in one component should have a minimal impact on the other components.

Notes The degree or strength of coupling between two components is measured by the amount and complexity of information transmitted between them.

Object oriented design has two types of coupling: interaction coupling and inheritance coupling.

1. ***Interaction coupling:*** Interaction coupling involves the amount and complexity of messages between components. It is good to have little interaction. The general guideline is to keep the message as simple and infrequent as possible. Objects connected to many complex messages are tightly coupled, meaning any change to one invariably leads to a ripple effect of changes in others.

   For types of coupling among objects or components, refer Table 13.1 as below:

Table 13.1: Types of Coupling among Objects or Components

Degree of Coupling	Name	Description
Very High	Content Coupling	Connection involves direct reference to attributes or methods of another object
High	Common Coupling	Connection involves two objects accessing a 'global data space', for both to read & write
Medium	Control Coupling	Connection involves explicit control of the processing logic of one object by another
Low	Stamp coupling	Connection involves passing an aggregate data structure to another object, which uses only a portion of the components of the data structure
Very low	Data coupling	Connection involves either simple data items or aggregate structures all of whose elements are used by the receiving object. (this is the goal of an architectural design)

Notes

2. ***Inheritance coupling:*** Inheritance coupling is a form of coupling between super and sub classes. A subclass is coupled to its superclass in terms of attributes and methods. We need high inheritance coupling. For this each specialisation class should not inherit lot of unrelated and unneeded methods and attributes.

Caution If the superclass is overwriting most of the methods or not using them, then it is an indication that the inheritance coupling is low.

Corollary 2: Single Purpose

Before studying corollary 2, we will discuss cohesion.

The interaction within a single object or software component is called cohesion. Cohesion reflects the "single-purposeness" of an object. Highly cohesive components can lower coupling because only a minimum of essential information need be passed between components. Method cohesion means that a method should carry one function.

A method that carries multiple functions is undesirable.

Did u know? Class cohesion means that all the class's methods and attributes must be highly cohesive, meaning to be used by internal methods or derived classes' methods.

Now we will discuss corollary 2 as follows:

- Every class should be clearly defined and necessary in the context of achieving the system's goals.
- When we document a class, we should be able to explain its purpose in a sentence or two.
- If we cannot, then the class should be subdivided into independent pieces.
- Each method must provide only one service.
- Each method should be of moderate size, no more than a page; half a page is better.

Corollary 3: Large Number of Simpler Classes

- There are benefits in having a large number of simpler classes because the chances of reusing smaller classes in other projects are high.
- Large and complex classes are too specialised to be reused.
- Object-oriented design offers a path for producing libraries of reusable parts.

Reusability is not used because of the following reasons:

- Software engineering textbooks teach new practitioners to build systems from "first principles"; reusability is not promoted or even discussed.
- The "not invented here" syndrome and the intellectual challenge of solving an interesting software problem in one's own unique way mitigates against reusing someone else's software component.
- Unsuccessful experiences with software reusability in the past have convinced many practitioners and development managers that the concept is not practical.

Notes

- Most organisations provide no reward for reusability; sometimes productivity is measured in terms of new lines of code written plus a discounted credit

Corollary 4: Strong Mapping

- A strong mapping links classes identified during analysis and classes designed during the design phase e.g., view and access classes.
- The analyst identifies objects' types and inheritance, and thinks about events that change the state of objects.
- The designer adds detail to this model perhaps designing screens, user interaction, and client-server interaction.

Corollary 5: Standardisation

- To reuse classes, we must have a good understanding of the classes.
- Most object-oriented systems come with several built-in class libraries.
- But these class libraries are not always well documented.
- Sometimes they are documented, but not updated.
- They must be easily searched, based on users' criteria.

Corollary 6: Designing with Inheritance

- When we implement a class, we have to determine its ancestor, what attributes it will have, and what messages it will understand.
- Then we have to construct its methods and protocols.
- Ideally, one has to choose inheritance to minimise the amount of program instructions.
- The primitive form of reuse is cut-and-paste reusability.

Now let us see the process of achieving multiple inheritance in a singe inheritance system:

- Single inheritance means that each class has only a single super class.
- The result of using a single inheritance hierarchy is the absence of ambiguity as to how an object will respond in a given method.
- We simply trace up the class tree beginning with the object's class, looking for a method of the same name.
- But languages like LISP or C++ have a multiple inheritance scheme whereby objects can inherit behavior from unrelated areas of the class tree.
- The complication here is how to determine which behavior to get from which class, particularly when several ancestors define the same method.
- One way of resolving this is to inherit from the most appropriate class and add an object of mother class as an attribute or aggregation. The other is to use the instance of the class (object) as an attribute.

Task Make distinction between stamp coupling and data coupling.

Notes

Self Assessment

Fill in the blanks:

1. An ..... is a fundamental truth that always is observed to be valid and for which there is no counter example or exception.
2. A ..... is a proposition that may not be self-evident but can be proved from accepted axioms.
3. A ..... is a proposition that follows from an axiom or another proposition that has been proven.
4. ..... is a measure of the strength of association established by a connection from one object or software component to another.
5. ..... is a binary relationship.
6. ..... coupling involves the amount and complexity of messages between components.
7. ..... coupling is a form of coupling between super and sub classes.
8. ..... means that all the class's methods and attributes must be highly cohesive, meaning to be used by internal methods or derived classes' methods.

13.2 Concept of Class Design

Object-oriented design requires taking the object identified during object-oriented analysis and designing classes to represent them.

As a class designer, we have to know the specifics of the class we are designing and also we should be aware of how that class interacts with other classes.

13.2.1 Object-oriented Design Philosophy

- Here one has to think in terms of classes. As new facts are acquired, we relate them to existing structures in our environment (model).
- After enough new facts are acquired about a certain area, we create new structures to accommodate the greater level of detail in our knowledge.
- The important activity in designing an application is coming up with a set of classes that work together to provide the functionality we desire.
- If we design the classes with reusability in mind, we will gain a lot of productivity and reduce the time for developing new applications.

13.2.2 Class Visibility: Designing well-defined Public, Private and Protected Protocols

In designing methods or attributes for classes, we are confronted with two problems. One is the protocol or interface to the class operations and its visibility and the other is how it is implemented. The class's protocol or the messages that a class understands, can be hidden from other objects (private protocol) or made available to other objects (public protocol).

Public protocols define the functionality and external messages of an object, while private protocols define the implementation of an object. It is important in object-oriented design to

Notes

define the public protocol between the associated classes in the application. This is a set of messages that a class of a certain generic type must understand, although the interpretation and implementation of each message is up to the individual class.

A class also might have a set of methods that it uses only internally, messages to itself. Thus, the private protocol (visibility) of the class includes messages that normally should not be sent from other objects; it is accessible only to operations of that class. In private protocol, only the class itself can use the method. The public protocol (visibility) defines the stated behavior of the class as a citizen in a population and is important information for users as well as future descendants, so it is accessible to all classes.

If the methods or attributes can be used by the class itself or its subclasses; a protected protocol can be used.

Did u know? In a protected protocol (visibility), subclasses can use the method in addition to the class itself.

Lack of a well-designed protocol can manifest itself as encapsulation leakage. The problem of encapsulation leakage occurs when details about a class's internal implementation are disclosed through the interface. As more internal details become visible, the flexibility to make changes in the future decreases. If an implementation is completely open, almost no flexibility is retained for future carefully controlled. However, do not make such a decision lightly because that could impact the flexibility and therefore the quality of the design.

Example: Public or protected methods that can access private attributes can reveal an important aspect of your implementation. If anyone uses these functions and you change their location, the type of attribute, or the protocol of the method, this could make the client application inoperable.

Design the interface between a superclass and its subclasses just as carefully as the class's interface to clients; this is the contract between the super- and subclasses. If this interface is not designed properly, it can lead to violating the encapsulation of the superclass. The protected portion of the class interface can be accessed only by subclasses. This feature is helpful but cannot express the totality of the relationship between a class and its subclasses.

Other important factors include which functions might or might not be overridden and how they must behave. It also is crucial to consider the relationship among methods. Some methods might need to be overridden in groups to preserve the class's semantics.

Caution Design your interface to subclasses so that a subclass that uses every supported aspect of that interface does not compromise the integrity of the public interface.

The following sections summarize the differences between these layers.

13.2.3 Private and Protected Protocol Layers: Internal

Items in these layers define the implementation of the object. Apply the design axioms and corollaries to decide what should be private: what attributes (instance variables)? What methods? Remember, highly cohesive objects can improve coupling because only a minimal amount of essential information need be passed between object.

Notes

13.2.4 Public Protocol Layer: External

Items in this layer define the functionality of the object. Here are some things to keep in mind when designing class protocols: Good design allows for polymorphism. Not all protocol should be public; again apply design axioms and corollaries 1.

Notes The following key questions must be answered: What are the class interfaces and protocols? What public (external) protocol will be used or what external messages must the system understand? What private or protected (internal) protocol will be used or what internal messages or messages from a subclass must the system understand?

Task Analyze the difference between protected protocol layer and public protocol layer.

13.2.5 Designing Classes: Refining Attributes

Attributes identified in object-oriented analysis must be refined with an eye on implementation during this phase. In the analysis phase, the name of the attribute is enough. But in the design phase, detailed information must be added to the model.

Attribute Types

The types of attribute are explained as below:

1. ***Single value attributes:*** Single value attributes consists of just one value or state.

Example: Name, address, salary.

2. ***Multiplicity or multi value attributes:*** Multiplicity or multi value attributes can have a collection of several values at a time.

Example:

(a) To list the students who have scored above 710 marks.

(b) If we want to keep record of the names of people who have called a customer support line for help.

3. ***Instance connection:*** It is reference to another object. It offers the mapping required by an object to accomplish its responsibilities.

Example: A person may have more than one account. A person has zero to many instance connections to Account(s). Similarly, an Account can be assigned to one or more person(s) (joint account). So an Account has zero to many instance connection to Person(s).

UML Attribute Presentation

Notes

The Attribute Presentation recommended by UML is shown as below:

Visibility name: Type – expression = initial – value

Where visibility may be

- + Public visibility
- # protected visibility
- - Private visibility

Type-expression is a language dependent specification of the implementation type of an attribute.

Initial value is a language-dependent expression for the initial value of a newly created object.

Example: + size: length = 100

Self Assessment

Fill in the blanks:

9. As a ....., we have to know the specifics of the class we are designing and also we should be aware of how that class interacts with other classes.
10. ..... protocols define the functionality and external messages of an object.
11. ..... protocols define the implementation of an object.
12. In private protocol, only the ..... itself can use the method.
13. Lack of a well-designed protocol can manifest itself as ..... leakage.
14. Items in public protocol layer define the ..... of the object.
15. ..... attributes can have a collection of several values at a time.

13.3 Summary

- An axiom is a fundamental truth that always is observed to be valid and for which there is no counter example or exception.
- A theorem is a proposition that may not be self-evident but can be proved from accepted axioms.
- Corollaries may be called Design rules, and all are derived from the two basic axioms.
- Coupling is a measure of the strength of association established by a connection from one object or software component to another.
- Interaction coupling involves the amount and complexity of messages between components. Inheritance coupling is a form of coupling between super and sub classes.
- Object-oriented design requires taking the object identified during object-oriented analysis and designing classes to represent them.
- As a class designer, we have to know the specifics of the class we are designing and also we should be aware of how that class interacts with other classes.

Notes

- Public protocols define the functionality and external messages of an object, while private protocols define the implementation of an object.

13.4 Keywords

Axiom: An axiom is a fundamental truth that always is observed to be valid and for which there is no counter example or exception.

Corollary: A corollary is a proposition that follows from an axiom or another proposition that has been proven.

Coupling: Coupling is a measure of the strength of association established by a connection from one object or software component to another.

Inheritance coupling: Inheritance coupling is a form of coupling between super-and sub-classes.

Interaction coupling: Interaction coupling involves the amount and complexity of messages between components.

Private protocols: Private protocols define the implementation of an object.

Public protocols: Public protocols define the functionality and external messages of an object.

Theorem: A theorem is a proposition that may not be self-evident but can be proved from accepted axioms.

13.5 Review Questions

1. What are the axioms of object-oriented design? Discuss.
2. What are corollaries? Draw the diagram to show origin of corollaries and explain.
3. Describe the concept of coupling.
4. Make distinction between interaction coupling and inheritance coupling.
5. Illustrate the concept of class design.
6. Elucidate the process of achieving multiple inheritance in a singe inheritance system.
7. Discuss the concept of public, private and protected protocols.
8. Analyze whether public protocol layer is external or internal.
9. Explain the concept of refining attributes.
10. What is an attribute presentation recommended by UML? Discuss.

Answers: Self Assessment

1.	Axiom	2.	Theorem
3.	corollary	4.	Coupling
5.	Coupling	6.	Interaction
7.	Inheritance	8.	Class cohesion
9.	class designer	10.	Public
11.	Private	12.	Class

Notes

13. encapsulation
14. Functionality
15. Multiplicity or multi value

13.6 Further Readings

Books

Booch, Grady, (1994), *Object-oriented Analysis & Design,* Addison Wesley

Laganiere, Robert, (2004), *Object-oriented Software Engineering,* TMH

Rumbaugh, J., (2007), *Object-oriented Modelling and Design with UML,* Pearson Education

Satzinger, (2007), *Object-oriented Analysis & Design with the Unified Process,* Thomson

Online links

http://developergeeks.com/article/40/ooad-identifying-classes-and-their-responsibilities

http://www.einsteincollege.ac.in/Assets/Department/Lecturer%20notes/CSE/UG/CS%201042%20OBJECT%20ORIENTED%20ANALYSIS%20AND %20DESIGN.pdf

http://www.slideshare.net/niitstudentcare/06-ooad-uml07

https://www.iit.edu/cpd/professional_learning/information_technology_cert/IT-T531.shtml

Notes

Unit 14: Steps for Class Design

CONTENTS

Objectives

Introduction

14.1 Steps of Class Design

14.1.1 Combining the Three Models to Obtain Operations on Classes

14.1.2 Designing Algorithms

14.1.3 Refactoring

14.1.4 Design Optimization

14.1.5 Implementation of Control

14.1.6 Adjustment of Inheritance

14.1.7 Design of Associations

14.1.8 Object Representation

14.1.9 Organizing Class Design

14.2 Documenting Design Decisions

14.3 Summary

14.4 Keywords

14.5 Review Questions

14.6 Further Readings

Objectives

After studying this unit, you will be able to:

- Define class design
- Discuss the steps included in class design
- Explain combining the three models to obtain operations on classes
- Describe the process of designing algorithms to implement operations
- Explain the process of optimizing access paths to data
- Implement control for external interactions
- Discuss adjustment of inheritance
- Discuss organizing class design

Introduction

The analysis model describes the information that the system must contain and the high-level operations that it must perform. The simplest and best approach is to carry the analysis classes directly into design. During design, choose among the different ways to realize the analysis

Notes

classes for minimizing execution time, memory, and other cost measures. New classes may be introduced to store intermediate results during program execution and avoid recomputation. OO design is an iterative process. During object design, the designer carries out the strategy chosen during the system design and fleshes out the details. There is a shift in emphasis from application domain concepts toward computer concepts. In this unit, we will discuss the steps included in class design.

14.1 Steps of Class Design

The objects discovered during analysis serve as the skeleton of the design, but the object designer must choose among different ways to implement them with an eye toward minimizing execution time, memory and other measures of cost.

Caution The operations identified during the analysis must be expressed as algorithms, with complex operations decomposed into simpler internal operations.

The classes, attributes and associations from analysis must be implemented as specific data structures. New object classes must be introduced to store intermediate results during program execution and to avoid the need for recomputation.

Notes Optimization of the design should not be carried to excess, as ease of implementation, maintainability, and extensibility are also important concerns.

During object design, the designer must perform the following steps:

1. Combining the three models to obtain operations on classes
2. Design algorithms to implement operations.
3. Refactoring
4. Optimize access paths to data.
5. Implement control for external interactions
6. Adjust class structure to increase inheritance.
7. Design associations.
8. Determine object representation.
9. Package classes and associations into modules

The steps included in class design are discussed below:

14.1.1 Combining the Three Models to Obtain Operations on Classes

After analysis, we have object, dynamic and functional model, but the object model is the main framework around which the design is constructed. The object model from analysis may not show operations. The designer must convert the actions and activities of the dynamic model and the processes of the functional model into operations attached to classes in the object model. Each state diagram describes the life history of an object. A transition is a change of state of the object and maps into an operation on the object. We can associate an operation with each event received by an object. In the state diagram, the action performed by a transition depends on both

Notes

the event and the state of the object. Therefore, the algorithm implementing an operation depends on the state of the object.

If the same event can be received by more than one state of an object, then the code implementing the algorithm must contain a case statement dependent on the state. An event sent by an object may represent an operation on another object. Events often occur in pairs, with the first event triggering an action and the second event returning the result on indicating the completion of the action. In this case, the event pair can be mapped into an operation performing the action and returning the control provided that the events are on a single thread. An action or activity initiated by a transition in a state diagram may expand into an entire data flow in the functional model.

Did u know? The network of processes within the data flow represents the body of an operation.

The flows in the diagram are intermediate values in operation. The designer converts the graphic structure of the diagram into linear sequence of steps in the algorithm. The process in the data flow represents sub operations. Some of them, but not necessarily all may be operations on the original target object or on other objects.

Determine the target object of a sub operation as follows:

- If a process extracts a value from input flow then input flow is the target.
- Process has input flow or output flow of the same type, input output flow is the target.
- Process constructs output value from several input flows, and then the operation is a class operation on output class.
- If a process has input or an output to data store or actor, data store or actor is the target.

14.1.2 Designing Algorithms

Each operation specified in the functional model must be formulated as an algorithm. The analysis specification tells what the operation does from the view point of its clients, but the algorithm shows how it is done. The analysis specification tells what the operation does from the view point of its clients, but the algorithm shows how it is done. An algorithm may be subdivided into calls on simpler operations, and so on recursively, until the lowest-level operations are simple enough to implement directly without refinement. The algorithm designer must decide on the following:

1. ***Choosing algorithms***: Many operations are simple enough that the specification in the functional model already constitutes a satisfactory algorithm because the description of what is done also shows how it is done. Many operations simply traverse paths in the object link network or retrieve or change attributes or links.

   Non trivial algorithm is needed for two reasons:

   (a) To implement functions for which no procedural specification

   (b) To optimize functions for which a simple but inefficient algorithm serves as a definition.

   Some functions are specified as declarative constraints without any procedural definition. In such cases, you must use your knowledge of the situation to invent an algorithm. The essence of most geometry problems is the discovery of appropriate algorithms and the proof that they are correct. Most functions have simple mathematical or procedural

Notes

definitions. Often the simple definition is also the best algorithm for computing the function or else is also so close to any other algorithm that any loss in efficiency is the worth the gain in clarity. In other cases, the simple definition of an operation would be hopelessly inefficient and must be implemented with a more efficient algorithm.

Example: Let us consider the algorithm for search operation. A search can be done in two ways like binary search (which performs log n comparisons on an average) and a linear search (which performs n/2 comparisons on an average). Suppose our search algorithm is implemented using linear search , which needs more comparisons. It would be better to implement the search with a much efficient algorithm like binary search.

Considerations in choosing among alternative algorithm include:

(a) *Computational Complexity:* It is essential to think about complexity i.e. how the execution time (memory) grows with the number of input values.

Example: For a bubble sort algorithm, time $\propto n^2$

Most other algorithms, time " n log n

(b) *Ease of implementation and understandability:* It is worth giving up some performance on non critical operations if they can be implemented quickly with a simple algorithm.

(c) *Flexibility:* Most programs will be extended sooner or later. A highly optimized algorithm often sacrifices readability and ease of change. One possibility is to provide two implementations of critical applications, a simple but inefficient algorithm that can be implemented quickly and used to validate the system, and a complicated but efficient algorithm whose correct implementation can be checked against the simple one.

(d) *Fine Timing the Object Model:* We have to think, whether there would be any alternatives, if the object model were structured differently.

2. ***Choosing Data Structures:*** Choosing algorithms involves choosing the data structures they work on. We must choose the form of data structures that will permit efficient algorithms. The data structures do not add information to the analysis model, but they organize it in a form convenient for the algorithms that use it.

3. ***Defining Internal Classes and Operations:*** During the expansion of algorithms, new classes of objects may be needed to hold intermediate results. New, low level operations may be invented during the decomposition of high level operations. A complex operation can be defined in terms of lower level operations on simpler objects. These lower level operations must be defined during object design because most of them are not externally visible. Some of these operations were found from "shopping –list". There is a need to add new internal operations as we expand high-level functions. When you reach this point during the design phase, you may have to add new classes that were not mentioned directly in the client's description of the problem. These low-level classes are the implementation elements out of which the application classes are built.

4. ***Assigning Responsibility for Operations:*** Many operations have obvious target objects, but some operations can be performed at several places in an algorithm, by one of the several places, as long as they eventually get done. Such operations are often part of a complex high-level operation with many consequences. Assigning responsibility for such operations can be frustrating, and they are easy to overlook in laying out object classes because they are easy to overlook in laying out object classes because they are not an

Notes

inherent part of one class. When a class is meaningful in the real world, then the operations on it are usually clear. During implementation, internal classes are introduced.

How do you decide what class owns an operation?

When only one object is involved in the operation, tell the object to perform the operation. When more than one object is involved, the designer must decide which object plays the lead role in the operation. For that, ask the following questions:

- Is one object acted on while the other object performs the action? It is best to associate the operation with the target of the operation, rather than the initiator.
- Is one object modified by the operation, while other objects are only queried for the information they contain? The object that is changed is the target.
- Looking at the classes and associations that are involved in the operation, which class is the most centrally-located in this subnetwork of the object model? If the classes and associations form a star about a single central class, it is the target of the operation.
- If the objects were not software, but the real world objects represented internally, what real world objects would you push, move, activate or manipulate to initiate operation?

Notes Assigning an operation within a generalization hierarchy can be difficult. Since the definitions of the subclasses within the hierarchy are often fluid and can be adjusted during design as convenient. It is common to move an operation up and down in the hierarchy during design, as its scope is adjusted.

14.1.3 Refactoring

Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure. It is a disciplined way to clean up code that minimizes the chances of introducing bugs. In essence, when you refactor, you are improving the design of the code after it has been written. "Improving the design after it has been written." That's an odd turn of phrase. In our current understanding of software development we believe that we design and then we code. A good design comes first, and the coding comes second. Over time the code will be modified, and the integrity of the system, its structure according to that design, gradually fades. The code slowly sinks from engineering to hacking.

Refactoring is the opposite of this practice. With refactoring you can take a bad design, chaos even, and rework it into well-designed code. Each step is simple, even simplistic. You move a field from one class to another, pull some code out of a method to make into its own method, and push some code up or down a hierarchy. Yet the cumulative effect of these small changes can radically improve the design. It is the exact reverse of the normal notion of software decay. With refactoring you find the balance of work changes. You find that design, rather than occurring all up front, occurs continuously during development. You learn from building the system how to improve the design. The resulting interaction leads to a program with a design that stays good as development continues.

Task Analyze the advantages of refactoring.

Notes

14.1.4 Design Optimization

The basic design model uses the analysis model as the framework for implementation. The analysis model captures the logical information about the system, while the design model must add details to support efficient information access. The inefficient but semantically correct analysis model can be optimized to make the implementation more efficient, but an optimized system is more obscure and less likely to be reusable in another context.

The designer must strike an appropriate balance between efficiency and clarity. During design optimization, the designer must:

1. ***Add Redundant Associations for Efficient Access:*** During analysis, it is undesirable to have redundancy in association network because redundant associations do not add any information. During design, however, we evaluate the structure of the object model for an implementation. For that, we have to answer the following questions:

   - Is there a specific arrangement of the network that would optimize critical aspects of the completed system?
   - Should the network be restructured by adding new associations?
   - Can existing associations be omitted?

     The associations that were useful during analysis may not form the most efficient network when the access patterns and relative frequencies of different kinds of access are considered. In cases where the number of hits from a query is low because only a fraction of objects satisfy the test, we can build an index to improve access to objects that must be frequently retrieved. Analyze the use of paths in the association network as follows:

   - Examine each operation and see what associations it must traverse to obtain its information. Note which associations are traversed in both directions, and which are traversed in a single direction only, the latter can be implemented efficiently with one way pointers.

   For each operation note the following items:

   - How often is the operation called? How costly is to perform?
   - What is the "fan-out" along a path through the network? Estimate the average count of each "many" association encountered along the path. Multiply the individual fan-outs to obtain the fan-out of the entire path; which represents the number of accesses on the last class in the path. Note that "one" links do not increase the fan-out, although they increase the cost of each operation slightly, don't worry about such small effects.
   - What is the fraction of "hits" on the final class, that is, objects that meets selection criteria (if any) and is operated on? If most objects are rejected during the traversal for some reason, then a simple nested loop may be inefficient at finding target objects. Provide indexes for frequent, costly operations with a low hit ratio because such operations are inefficient to implement using nested loops to traverse a path in the network.

2. ***Rearranging Execution Order for Efficiency:*** After adjusting the structure of the object model to optimize frequent traversal, the next thing to optimize is the algorithm itself. Algorithms and data structures are directly related to each other, but we find that usually the data structure should be considered first. One key to algorithm optimization is to eliminate dead paths as early as possible. Sometimes the execution order of a loop must be inverted.

Notes

3. ***Saving Derived Attributes to Avoid Recomputation:*** Data that is redundant because it can be derived from other data can be "cached" or store in its computed form to avoid the overhead of recomputing it. The class that contains the cached data must be updated if any of the objects that it depends on are changed.

   Derived attributes must be updated when base values change. There are three ways to recognise when an update is needed:

   - *Explicit update:* Each attribute is defined in terms of one or more fundamental base objects. The designer determines which derived attributes are affected by each change to a fundamental attribute and inserts code into the update operation on the base object to explicitly update the derived attributes that depend on it.
   - *Periodic Recomputation:* Base values are updated in bunches. Recompute all derived attributes periodically without recomputing derived attributes after each base value is changed. Recomputation of all derived attributes can be more efficient than incremental update because some derived attributes may depend on several base attributes and might be updated more than once by incremental approach. Periodic recomputation is simpler than explicit update and less prone to bugs. On the other hand, if the data set changes incrementally a few objects at a time, periodic recomputation is not practical because too many derived attributes must be recomputed when only a few are affected.
   - *Active values:* An active value is a value that has dependent values. Each dependent value registers itself with the active value, which contains a set of dependent values and update operations. An operation to update the base value triggers updates all dependent values, but the calling code need not explicitly invoke the updates. It provides modularity.

14.1.5 Implementation of Control

The designer must refine the strategy for implementing the state – event models present in the dynamic model. As part of system design, you will have chosen a basic strategy for realizing dynamic model, during object design flesh out this strategy. There are three basic approaches to implementing the dynamic model:

1. ***State as Location within a Program:*** This is the traditional approach to representing control within a program. The location of control within a program implicitly defines the program state. Any finite state machine can be implemented as a program. Each state transition corresponds to an input statement. After input is read, the program branches depending on the input event received. Each input statement need to handle any input value that could be received at that point. In highly nested procedural code, low-level procedures must accept inputs that they may know nothing about and pass them up through many levels of procedure calls until some procedure is prepared to handle them. One technique of converting state diagram to code is as follows:

   (a) Identify the main control path. Beginning with the initial state, identify a path through the diagram that corresponds to the normally expected sequence of events. Write the name of states along this path as a linear sequence of events. Write the names of states along this path as a linear sequence. This becomes a sequence of statements in the program.

   (b) Identify alternate paths that branch off the main path and rejoin it later. These become conditional statements in the program.

Notes

(c) Identify backward paths that branch off the main loop and rejoin it earlier. These become loops in program. If multiple backward paths that do not cross, they become nested loops. Backward paths that cross do not nest and can be implemented with goto if all else fails, but these are rare.

(d) The status and transitions that remain correspond to exception conditions. They can be handled using error subroutines, exception handling supported by the language, or setting and testing of status flags. In the case of exception handling, use goto statements.

2. ***State machine engine:*** The most direct approach to control is to have some way of explicitly representing and executing state machine.

Example: State machine engine class helps execute state machine represented by a table of transitions and actions provided by the application.

Each object instance would contain its own independent state variables but would call on the state engine to determine next state and action. This approach allows you to quickly progress from analysis model to skeleton prototype of the system by defining classes from object model state machine and from dynamic model and creating "stubs" of action routines. A stub is a minimal definition of function/subroutine without any internal code. Thus if each stub prints out its name, technique allows you to execute skeleton application to verify that basic flow of control is correct. This technique is not so difficult.

3. ***Control as Concurrent Tasks:*** An object can be implemented as task in programming language/operating system. It preserves inherent concurrency of real objects. Events are implemented as inter task calls using facilities of language/operating system. Concurrent C++/Concurrent Pascal support concurrency. Major Object Oriented languages do not support concurrency.

14.1.6 Adjustment of Inheritance

The definitions of classes and operations can often be adjusted to increase the amount of inheritance.

The designer should:

1. ***Rearrange classes and operations:*** Sometimes the same operation is defined across several classes and can easily be inherited from a common ancestor, but more often operations in different classes are similar but not identical. By slightly modifying the definitions of the operations or the classes, the operations can often be made to match so that they can be covered by a single inherited operation. Before inheritance can be used, each operation must have the same interface and the types of arguments and results. If the signatures match, then the operations must be examined to see if they have the same semantics. The following kinds of adjustments can be used to increase the chance of inheritance.

   - Some operations may have fewer arguments than others. The missing arguments can be added but ignored.
   - Some operations may have few arguments because they are special cases of more general arguments. Implement the special operations by calling the general operation with appropriate parameter values.
   - Similar attributes in different classes may have different names. Give the attributes the same name and move them to a common ancestor class. These operations that access the attributes will match better.

Notes

- Any operation may be defined on several different classes in a group but undefined on the other classes. Define it on the common ancestor class and define it as no operation on the values that do not care about it.

2. ***Abstracting Out Common Behavior:*** Reexamine the object model looking for commonality between classes. New classes and operations are often added during design. If a set of operations/attributes seems to be repeated in two classes, it is possible that the two classes are specialized variations of the same thing. When common behavior has been recognized, a common superclass can be created that implements the shared features, specialized features in subclass. This transformation of the object model is called abstracting out a common superclass/common behavior usually the superclass is abstract meaning no direct instances. Sometimes a superclass is abstracted even when there is only one subclass; here there is no need of sharing. Superclass may be reusable in future projects. It is an addition to the class library. When a project is completed, the reusable classes should be collected, documented and generalized so that they may be used in future projects. Another advantage of abstract superclasses other than sharing and reuse is modularity.

Did u know? Abstract superclasses improve the extensibility of a software product and helps in the configuration management of software maintenance and distribution.

3. ***Use Delegation to Share Implementation:*** Sometimes programmers use inheritance as an implementation technique with no intention of guaranteeing the same behavior. Sometimes an existing class implements some of the behavior that we want to provide in a newly defined class, although in other respects the two classes are different. The designer may inherit from the existing class to achieve part of the implementation of the new class. This can lead to problems –unwanted behavior.

14.1.7 Design of Associations

During object design phase, we must formulate a strategy for implementing all associations in the object model. We can either choose a global strategy for implementing all associations uniformly, or a particular technique for each association.

Analyzing Association Traversal

Associations are inherently bidirectional. If association in your application is traversed in one direction, their implementation can be simplified. The requirements on your application may change; you may need to add a new operation later that needs to traverse the association in reverse direction. For prototype work, use bidirectional association so that we can add new behavior and expand/modify. In the case of optimization work, optimize some associations.

One-way association

- If an association is only traversed in one direction it may be implemented as pointer.
- If multiplicity is “many” then it is implemented as a set of pointers.
- If the “many” is ordered, use list instead of set.
- A qualified association with multiplicity one is implemented as a dictionary object (A dictionary is a set of value pairs that maps selector values into target values).
- Qualified association with multiplicity “many” are rare. (It is implemented as dictionary set of objects).

Notes

Two-way Associations

Many associations are traversed in both directions, although not usually with equal frequency. There are three approaches to their implementation:

- Implement as an attribute in one direction only and perform a search when a backward traversal is required. This approach is useful only if there is great disparity in traversal frequency and minimizing both the storage cost and update cost are important.
- Implement as attributes in both directions. It permits fast access, but if either attribute is updated then the other attribute must also be updated to keep the link consistent. This approach is useful if accesses outnumber updates.
- Implement as a distinct association object independent of either class. An association object is a set of pairs of associated objects stored in a single variable size object. An association object can be implemented using two dictionary object one for forward direction and other for reverse direction.

Link Attributes

Its implementation depends on multiplicity.

- If it is a one-one association, link attribute is stored in any one of the classes involved.
- If it is a many-one association, the link attribute can be stored as attributes of many object, since each "many object appears only once in the association.
- If it is a many-many association, the link attribute can't be associated with either object; implement association as distinct class where each instance is one link and its attributes.

14.1.8 Object Representation

Implementing objects is mostly straight forward, but the designer must choose when to use primitive types in representing objects and when to combine groups of related objects. Classes can be defined in terms of other classes, but eventually everything must be implemented in terms of built-in-primitive data types, such as integer strings, and enumerated types.

Example: Consider the implementation of a social security number within an employee object. It can be implemented as an attribute or a separate class.

Defining a new class is more flexible but often introduces unnecessary indirection. In a similar vein, the designer must often choose whether to combine groups of related objects.

14.1.9 Organizing Class Design

Programs consist of discrete physical units that can be edited, compiled, imported, or otherwise manipulated. In C and Fortran the units are source files; In Ada, it is packages. In object oriented languages, there are various degrees of packaging.

Caution In any large project, careful partitioning of an implementation into packages is important to permit different persons to cooperatively work on a program.

Notes

Packaging involves the following issues:

- ***Hiding internal information from outside view:*** One design goal is to treat classes as black boxes, whose external interface is public but whose internal details are hidden from view. Hiding internal information permits implementation of a class to be changed without requiring any clients of the class to modify code. Additions and changes to the class are surrounded by "fire walls" that limit the effects of any change so that changes can be understood clearly. Trade off between information hiding and optimization activities. During analysis, we are concerned with information hiding. During design, the public interface of each class must be defined carefully. The designer must decide which attributes should be accessible from outside the class. These decisions should be recorded in the object model by adding the annotation {private} after attributes that are to be hidden, or by separating the list of attributes into 2 parts. Taken to an extreme a method on a class could traverse all the associations of the object model to locate and access another object in the system .This is appropriate during analysis, but methods that know too much about the entire model are fragile because any change in representation invalidates them. During design we try to limit the scope of any one method. We need to define the bounds of visibility that each method requires. Specifying what other classes a method can see defines the dependencies between classes. Each operation should have a limited knowledge of the entire model, including the structure of classes, associations and operations. The fewer things that an operation knows about, the less likely it will be affected by any changes. The fewer operations know about details of a class, the easier the class can be changed if needed.

  The following design principles help to limit the scope of knowledge of any operation:

  - Allocate to each class the responsibility of performing operations and providing information that pertains to it.
  - Call an operation to access attributes belonging to an object of another class
  - Avoid traversing associations that are not connected to the current class.
  - Define interfaces at as high a level of abstraction as possible.
  - Hide external objects at the system boundary by defining abstract interface classes, that is, classes that mediate between the system and the raw external objects.
  - Avoid applying a method to the result of another method, unless the result class is already a supplier of methods to the caller. Instead consider writing a method to combine the two operations.

- ***Coherence of entities:*** One important design principle is coherence of entities. An entity, such as a class, an operation, or a module, is coherent if it is organized on a consistent plan and all its parts fit together toward a common goal. It shouldn't be a collection of unrelated parts. A method should do one thing well .a single method should not contain both policy and implementation. "A policy is the making of context dependent decisions." "Implementation is the execution of fully specified algorithms."

  Policy involves making decisions, gathering global information, interacting with outside world and interpreting special cases. Policy methods contain input output statements, conditionals and accesses data stores. It doesn't contain complicated algorithms but instead calls various implementation methods. An implementation method does exactly one operation without making any decisions, assumptions, defaults or deviations. All information is supplied as arguments (list is long). Separating policy and implementation increase reusability. Therefore implementation methods don't contain any context dependency. So they are likely to be reusable Policy method need to be rewritten in an

application , they are simple and consists of high level decisions and calls on low-level methods. A class shouldn't serve too many purposes.

- ***Constructing physical modules***
  - ❖ During analysis and system design phases we partitioned the object model into modules.
  - ❖ The initial organization may not be suitable for final packaging of system implementation new classes added to existing module or layer or separate module. Modules should be defined so that interfaces are minimal and well defined. Connectivity of object model can be used as a guide for partitioning modules. Classes that are closely connected by associations should be in the same module. Loosely connected classes should be grouped in separate modules. Classes in a module should represent similar kinds of things in the application or should be components of the same composite object. Try to encapsulate strong coupling within a module. Coupling is measured by number of different operations that traverse a given association. The number expresses the number of different ways the association is used, not the frequency.

Task When an entity is said to be coherent? Discuss.

Self Assessment

Fill in the blanks:

1. The ..... works to implement the objects discovered during analysis phase.
2. The objects discovered during ..... serve as the skeleton of the design.
3. The operations identified during the analysis must be expressed as .....
4. The classes, attributes and associations from analysis must be implemented as specific .....
5. An ..... sent by an object may represent an operation on another object.
6. Complex operation can be defined in terms of ..... operations on simpler objects.
7. ..... is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure.
8. The class that contains the ..... data must be updated if any of the objects that it depends on are changed.
9. An ..... value is a value that has dependent values.
10. A ..... is a minimal definition of function /subroutine without any internal code.
11. When a project is completed, the ..... classes should be collected, documented and generalized so that they may be used in future projects.
12. If ..... in your application is traversed in one direction, their implementation can be simplified.
13. Hiding ..... information permits implementation of a class to be changed without requiring any clients of the class to modify code.

Notes

14.2 Documenting Design Decisions

The above design decisions must be documented when they are made, or you will become confused. This is especially true if you are working with other developers. It is impossible to remember design details for any non trivial software system, and documentation is the best way of transmitting the design to others and recording it for reference during maintenance.

The design document is an extension of the Requirements Analysis Document.

- The design document includes revised and much more detailed description of the object model-both graphical and textual. Additional notation is appropriate for showing implementation decisions.

Example: Arrows showing the traversal direction of associations and pointers from attributes to other objects.

- Functional model will also be extended. It specifies all operation interfaces by giving their arguments, results, input-output mappings and side effects.
- Dynamic model – if it is implemented using explicit state control or concurrent tasks then the analysis model or its extension is adequate. If it is implemented by location within program code, then structured pseudocode for algorithms is needed.

Keep the design document different from analysis document .The design document includes many optimizations and implementation artifacts. It helps in validation of software and for reference during maintenance. Traceability from an element in analysis to element in design document should be straightforward. Therefore the design document is an evolution of analysis model and retains same names.

Self Assessment

Fill in the blanks:

14. The ..... document is an extension of the Requirements Analysis Document.
15. If ..... model is implemented using explicit state control or concurrent tasks then the analysis model or its extension is adequate.

Case Study

Refining Attributes for the ViaNet Bank Objects

Here, we go through the ViaNet bank ATM system classes and refine the attributes as discussed below:

Refining Attributes for the Bank Client Class

Attributes for the Bank Client Class include:

firstName

lastName

pinNumber

cardNumber

Contd...

Notes

At this stage, we need to add more information to these attributes, such as visibility and implementation type. Furthermore, additional attributes can be identified to enable implementation of the class:

#firstName: String

#lastName: String

#pinNumber: String

#cardNumber: String

#account: Account (instance connection)

To design the association between the BankClient and the Account classes, we need to add an account attribute of type Account, since the BankClient needs to know about his or her account and this attribute can provide such information for the BankClient class. This is an example of instance connection, where it represents the association between the BankClient and the Account objects. All the attributes have been given protected visibility.

Refining Attributes for the Account Class

Here is the refined list of attributes for the Account class:

#number: String

#balance: float

#transaction: Transaction (This attribute is needed for implementing the association between the Account and Transaction classes.)

#bankClient: BankClient (This attribute is needed for implementing the association between the Account and BankClient classes.)

At this point we .must make the Account class very general, so that it can be reused by the checking and savings accounts.

Refining Attribute for the Transaction Class

The attributes for the Transaction class are these:

#transID: String

#transDate: Date

#transTime: Time

#transType: String

#amount: float

#postBalance: float

Refining Attributes for the ATM Machine Class

The ATM Machine class could have the following attributes:

#address: String

#state: String

Refining Attributes for the CheckingAccount Class

Add the savings attribute to the class. The purpose of this attribute is to implement the association between the CheckingAccount and SavingsAccount classes.

Contd...

Notes

Refining Attributes for the SavlngsAccount Class

Add the *checking* attribute to the class. The purpose of this attribute is to implement the association between the SavingsAccount and CheckingAccount classes.

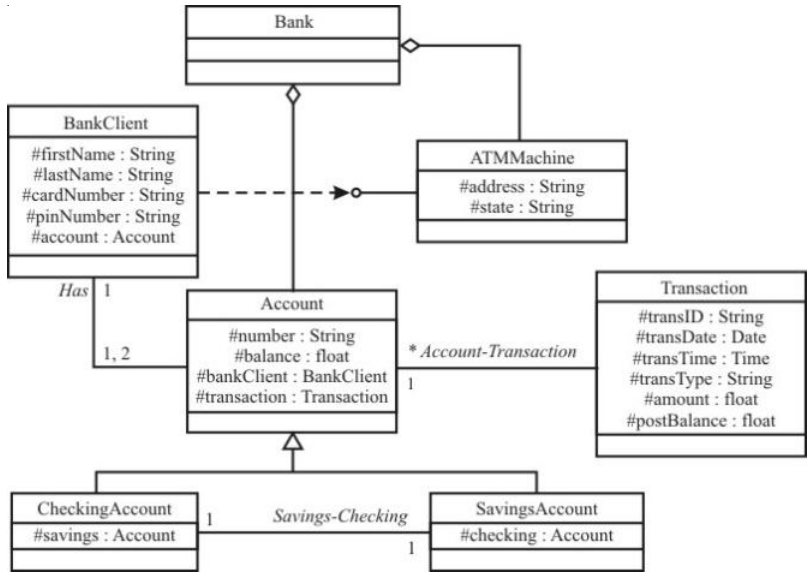
Figure 1 shows a more complete UML class diagram for the bank system. At this stage, we also need to add a very short description of each attribute or certain attribute constraints. For example,

Class ATM Machine

#address: String (The address for this ATM machine.)

#state: String (The state of operation for this ATM machine, such as running, off, idle, out of money, security alarm.)

Figure 1: A more Complete UML Class Diagram for the ViaNet Bank System



Question

Why do we not need the account attribute for the Transaction class? Hint: Do transaction objects need to know about account objects?

Source: Bahrami, Ali, *"Object Oriented Systems Development,"* Tata McGraw-Hill Education

14.3 Summary

- The class design phase determines the full definitions of the classes and associations used in the implementation, as well as the interfaces and algorithms of the methods used to implement operations.
- After analysis, we have object, dynamic and functional model, but the object model is the main framework around which the design is constructed.
- An algorithm may be subdivided into calls on simpler operations, and so on recursively, until the lowest-level operations are simple enough to implement directly without refinement.

Notes

- Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure.
- The basic design model uses the analysis model as the framework for implementation. The analysis model captures the logical information about the system, while the design model must add details to support efficient information access.
- The designer must refine the strategy for implementing the state – event models present in the dynamic model.
- The definitions of classes and operations can often be adjusted to increase the amount of inheritance.
- If association in your application is traversed in one direction, their implementation can be simplified.
- Implementing objects is mostly straight forward, but the designer must choose when to use primitive types in representing objects and when to combine groups of related objects.
- It is impossible to remember design details for any non trivial software system, and documentation is the best way of transmitting the design to others and recording it for reference during maintenance.

14.4 Keywords

Active value: An active value is a value that has dependent values.

***Class design*:** The class design phase determines the full definitions of the classes and associations used in the implementation, as well as the interfaces and algorithms of the methods used to implement operations.

Dictionary: A dictionary is a set of value pairs that maps selector values into target values.

Implementation: Implementation is the execution of fully specified algorithms.

Policy: A policy is the making of context dependent decisions.

Refactoring: Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code; yet improves its internal structure.

Stub: A stub is a minimal definition of function/subroutine without any internal code.

Transition: A transition is a change of state of the object and maps into an operation on the object.

14.5 Review Questions

1. Discuss the concept of class design.
2. "In the state diagram, the action performed by a transition depends on both the event and the state of the object." Comment.
3. Illustrate the steps taken for designing algorithms.
4. How does refactoring assists in improving the design of the code? Discuss.
5. Describe the steps taken by the designer during design optimization.
6. Elucidate the basic approaches used for implementing the dynamic model.

Notes

7. Illustrate the process of increasing the amount of inheritance.
8. Make distinction between one-way association and two-way association.
9. Explain the issues included in packaging.
10. Discuss the concept of documenting design decisions.

Answers: Self Assessment

1.	object-designer	2.	analysis
3.	algorithms	4.	data structures
5.	event	6.	lower level
7.	Refactoring	8.	Cached
9.	active	10.	Stub
11.	reusable	12.	Association
13.	internal	14.	Design
15.	dynamic		

14.6 Further Readings

Books

Booch, Grady, (1994), *Object-oriented Analysis & Design,* Addison Wesley

Laganiere, Robert, (2004), *Object-oriented Software Engineering,* TMH

Rumbaugh, J., (2007), *Object-oriented Modelling and Design with UML,* Pearson Education

Satzinger, (2007), *Object-oriented Analysis & Design with the Unified Process,* Thomson

Online links

http://web.engr.oregonstate.edu/~budd/Books/oopintro3e/info/chap03.pdf

http://www.mamcet.com/it/e-learning/5sem/OOAD/OOAD-2-MARKS.pdf

http://www.perflensburg.net/cp-web/djruobde.htm

http://www.trainingetc.com/PDF/TE1802eval.pdf