UNIT-II

CHAPTER- 4

OBJECT ORIENTED METHODOLOGIES

OBJECTIVES:

At the end of this chapter, students should be able to

- Define and understand the software development process
- Describe the object-oriented systems development
- List and explain the object-oriented methodologies

INTRODUCTION:

Basic Definitions

- A methodology is explained as the science of methods.
- A method is a set of procedures in which a specific goal is approached step by step.
- 1986: Booch came up with the object-oriented design concept, the Booch method.
- 1987: Sally Shlaer and Steve Mellor came up with the concept of the recursive design approach.
- 1989: Beck and Cunningham came up with class-responsibility-collaboration (CRC) cards.
• 1990: Wirfs-Brock, Wilkerson, and Wiener came up with responsibility-driven design.

• 1991: Peter Coad and Ed Yourdon developed the Coad lightweight and prototype-oriented approach.

• 1991: Jim Rumbaugh led a team at the research labs of General Electric to develop the object modeling technique (OMT).

• 1994: Ivar Jacobson introduced the concept of the use case.

Object-Oriented Systems Development: Using the Unified Modeling Language

Goals

• Object-Oriented Methodologies
  – The Rumbaugh et al. OMT
  – The Booch methodology
  – Jacobson's methodologies

• Unified Approach (UA)

• layered Architecture
Survey of Some of the Object-Oriented Methodologies

- Many methodologies are available to choose from for system development.
- Here, we look at the methodologies developed by Rumbaugh et al., Booch, and Jacobson which are the origins of the Unified Modeling Language (UML) and the bases of the UA.

**Rumbaugh et. al.’s Object Modeling Technique (OMT)**

- OMT describes a method for the analysis, design, and implementation of a system using an object-oriented technique.
- OMT consists of four phases, which can be performed iteratively:
  - 1. *Analysis*. The results are objects and dynamic and functional models.
  - 2. *System design*. The result is a structure of the basic architecture of the system.
  - 3. *Object design*. This phase produces a design document, consisting of detailed objects and dynamic and functional models.
  - 4. *Implementation*. This activity produces reusable, extendible, and robust code.
OMT Modeling

- OMT separates modeling into three different parts:
  - 1. An object model, presented by the object model and the data dictionary.
  - 2. A dynamic model, presented by the state diagrams and event flow diagrams.

Object Model

- Describes structure of objects in system
  - Identity
  - Relationship to other objects
  - Attributes
  - Operations

- Represented graphically with object diagram
  - Contains classes interconnected by association lines
  - Each class represents a set of individual objects
  - Association line represents set of links from objects of one class to objects of another class
OMT Object Model of a Bank System

![Diagram of OMT Object Model]

FIGURE 4-1
The OMT object model of a bank system. The boxes represent classes and the filled triangle represents specialization. Association between Account and transaction is one to many; since one account can have many transactions, the filled circle represents many (zero or more). The relationship between Client and Account classes is one to one: A client can have only one account and account can belong to only one person (in this model joint accounts are not allowed).

OMT Dynamic Model

- OMT state transition diagram is a network of states & events
  - Depict states, transitions, events, actions
  - Each state receives one or more events at which time it makes the transition to the next state
  - Next state depends on events + current state
OMT Functional Model

- Describes business process without focusing on computer systems details
- OMT data flow diagram shows flow of data between different processes in business
  - Processes – function performed
  - Data flow – direction of data element movement
  - Data store – location where data are stored
  - External entity – source/destination of data element

**FIGURE 4–2**
State transition diagram for the bank application user interface. The round boxes represent states and the arrows represent transitions.
OMT DFD of ATM System

Booch Methodology

- Analysis & design method using object oriented paradigm
  - Start with class & object diagrams in analysis phase
  - Diagrams refined in various steps
  - Add design symbols when ready to generate code
    - Object-oriented code documented
Diagrams in Booch Method

- Class diagrams
- Object diagrams
- State transition diagrams
- Module diagrams
- Process diagrams
- Interaction diagrams

Eg. of Booch class diagram

**FIGURE 4-4**
Object modeling using Booch notation. The arrows represent specialization; for example, the class Taurus is subclass of the class Ford.
Eg. of Booch state transition diagram

**FIGURE 4–5**
An alarm class state transition diagram with Booch notation. This diagram can capture the state of a class based on a stimulus. For example, a stimulus causes the class to perform some processing, followed by a transition to another state. In this case, the alarm silenced state can be changed to alarm sounding state and vice versa.
**OO Methodologies**

- Rumbaugh et al.’s Object Modeling Technique
  - Jim Rumbaugh & his colleagues developed the Object Modeling Technique (OMT) for analysis, design, and implementation of a system using an OO technique
  - OMT is a fast, intuitive approach for identifying & modeling all the objects making up a system
  - Consists of 4 phase, which can be performed iteratively:
1. Analysis ==> results are objects, and dynamic and functional models

2. System design ==> results are a structure of the basis architecture of the system along with high-level strategy decisions

3. Object design ==> produces a design document

4. Implementation ==> produces reusable, extendible, & robust code

- OMT separates modeling into 3 different parts:
  - **Object Model** – presented by object model & data dictionary
  - **Dynamic model** – presented by state diagrams & event flow diagrams
  - **Functional Model** – presented by data flow & constraints

- **Booch Methodology**
  - widely used OO method
  - covers the analysis & design phases of an OO system
  - Booch is criticized for his large set of symbols
• Booch method consists of those diagrams ==> class diagram, object diagrams, state transition diagrams, module diagrams, process diagrams & interaction diagrams

• Encompases both a “micro development process” and a “macro development process”

• The micro development process consists of the following steps:
  - Identify classes & objects
  - Identify class & object semantics
  - Identify class & object relationship
  - Identify class & object interfaces & implementation

• The macro development process serves as controlling framework for the micro process. Consists of the following steps:
  - Conceptualization
  - Analysis and development of the model
  - Design or create the system architecture
  - Evolution or implementation
  - Maintenance
The Jacobson et al. methodologies

- covers the entire life cycle & stress traceability between different phases
- The heart of their methodologies is the Use-Case concept
- **OOSE : Objectory**
  - Is a OO dev. with the specific aim to fit the development of large, real-time systems
  - Development process also called *Use-Case Driven Development*

- **OOBE**
  - Is object modeling at the enterprise level
  - Consists of the following phases:
    - Analysis phase
    - Design & implementation phases
    - Testing phase
Booch’s Macro Development Process

• Conceptualization
  – Establish core requirements
  – Establish goal
  – Develop prototype to prove concepts

• Analysis & development of model
  – Use class diagram to describe roles & responsibilities of objects in system
  – Use object diagram or interaction diagram to describe desired behavior of system in terms of scenarios

• Design & create system architecture
  – Use class diagram to decide what classes exist, how they relate to one another
  – Use object diagram to decide what mechanisms are used to regulate how objects collaborate
  – Use module diagram to map where each class & object should be declared
  – Use process diagram to determine process allocation to processors
  – Determine schedule for multiple processes on processors
• Evolution or Implementation
  – Refine system through iterations
  – Produce stream of software implementations/executable releases

• Maintenance
  – Make localized changes to add new requirements/eliminate bugs

**Booch’s Micro Development Process**

• Day to day activities for each macro development process

• Can consists of following steps
  – Identify classes & objects
  – Identify Class & object semantics
  – Identify class & object relationships
  – Identify class & object interfaces/implementation

**Jacobson et al. Methodologies**

• Object-oriented business engineering (OOBE)
• Object-oriented software engineering (OOSE)
• Object Factory for Software Development (Objectory)
• Cover entire life cycle
• Stress traceability between phases – backward & forward
• Enables reuse of analysis & design work
• Use-case concepts used
A Unified Approach (UA)

- the work done by Booch, Rumbaugh, and Jacobson ==> attempt to unify their modeling efforts
- UA (refer to figure 1 ) establishes a unifying & unitary framework around their works by utilizing the UML
- UML ==> to describe model, & document the SW dev. process
- Not to introduce another methodology
- To combine the best practices, processes, methodologies, & guidelines along with UML notations & diagrams ==> for better understanding OO concepts & systems development
- UA to SW dev. revolves around (but is not limited to ) the following processes & concepts (refer 4.8). The processes are:
  - Use-case driven development
  - OO analysis
  - OO design
  - Incremental development & prototyping
  - Continuous testing
- Methods & technology employed include:
  - UML used for modeling
  - Layered Approach
  - Repository for OO sys. Dev. patterns & frameworks
  - Components-based dev.
SUMMARY

OMT describes a method for the analysis, design, and implementation of a system using an object-oriented technique. OMT separates modeling into three different parts: An object model, presented by the object model and the data dictionary. A dynamic model, presented by the state diagrams and event flow diagrams. A functional model, presented by data flow and constraints. The unified approach consists of the following processes:

- Use-case driven development
- OO analysis
- OO design
- Incremental development & prototyping
- Continuous testing

KEY TERMS:

OMT
Object model
Dynamic model
Object Model
OMT Functional Model
Framework

KEY TERM QUIZ

1. **OMT** describes a method for the analysis, design, and implementation of a system using an object-oriented technique.
2. An **object model** is presented by the object model and the data dictionary.
3. A **dynamic model** is presented by the state diagrams and event flow diagrams.
4. A **functional model** is presented by data flow and constraints.
5. **Object Model** describes structure of objects in system
6. **OMT Functional Model** describes business process without focusing on computer systems details
7. A **framework** is a way of presenting a generic solution to a problem that can be applied to all levels in a development.
MULTIPLE CHOICE

1. **OMT** describes a method for the analysis, design, and implementation of a system using an object-oriented technique.
   (a) OMT (object modeling technique) (b) object model
2. An **object model** is presented by the object model and the data dictionary.
   (a) Dynamic (b) object model
3. A **dynamic model** is presented by the state diagrams and event flow diagrams.
   (a) Dynamic (b) object model
4. A **functional model** is presented by data flow and constraints.
   (a) Dynamic (b) object model (c) function model
5. **Object Model** describes structure of objects in system
   (a) Dynamic (b) object model (c) function model
6. **OMT Functional Model** describes business process without focusing on computer systems details
   (a) Dynamic (b) object model (c) OMT function model
7. A **framework** is a way of presenting a generic solution to a problem that can be applied to all levels in a development.
   (a) Framework (b) object model (c) pattern

REVIEW QUESTIONS:

1. What are the phases of OMT? Briefly describe each phase (Pg 63)
2. What is an object model? What are the OMT models? (Pg 63)
3. What is dynamic model? What are the OMT models? (Pg 63)
4. What is functional model? What are the OMT models? (Pg 64)
5. What is the main advantage of DFD? (Pg 64)
6. What is the strength of OMT? (Pg 63-64)
7. Name five Booch diagrams (Pg 65)
8. Briefly describe the Booch system development process (Pg 66-67)
9. What is the strength of Booch methodology? (Pg 65)
10. What is Objection? (Pg 68)
11. Name the models in Objectory (Pg 70)
12. What is a use case? (Pg 68)
13. What is the reason for having abstract use cases? (Pg 69)
14. Name some ways of describing use cases (Pg 69)
15. What must a use case contain? (Pg 69)
16. What is the strength of Jacobson methodology? (Pg 68)
17. Differentiate between patterns and frameworks (Pg 78)
18. Define Pattern (Pg 72)
19. Define Proto Pattern (Pg 73)
20. Name the characteristics of a good pattern (Pg 73)
21. Differentiate between generative and non-generative patterns (Pg 73)
22. List out the components of a Pattern template (Pg 74)
23. What is a pattern thumbnail? (Pg 76)
24. What are Anti Patterns? (Pg 76)
25. Name the guidelines for capturing patterns (Pg 76)
26. What is pattern mining? (Pg 76)
27. Define framework (Pg 77)
28. What is Unified Approach? (Pg 78)
29. Give the steps in OOA (Pg 80)
30. Give the steps in OOD (Pg 80)
31. What is layered approach to software development? (Pg 82)
32. Give the significance of Business Layer (Pg 83)
33. Give the significance of UI(View) Layer (Pg 84)
34. Give the significance of Access Layer (Pg 84)

Reference:
CHAPTER- 5
UNIFIED MODELING LANGUAGE

Chapter objectives

You should be able to define and understand

Modeling and its benefits

Different types of model

Basics of **UNIFIED MODELING LANGUAGE (UML)** and its modeling diagrams.

UML Class diagrams
UML Use case diagrams
UML Sequence diagrams
UML Collaboration diagrams
UML State chart diagrams
UML Activity diagrams
UML Component diagrams
UML Deployment diagrams

INTRODUCTION

A model is an abstract representation of a system, constructed to understand the system prior to building or modifying it. It is a model of a simplified representation of reality. Models can represent static or dynamic situations. The unified modeling language (UML) is a language for specifying, constructing, visualizing and documenting the software system and its components.
STATIC AND DYNAMIC MODELS

A model is an abstract representation of a system, constructed to understand the system prior to building or modifying it. It is a model of a simplified representation of reality. Models can represent static or dynamic situations.

Static model
- It can be viewed as a snapshot of a system’s parameters at rest or a specific point in time. They are needed to represent the structural or static aspect of a system. The UML class diagram is an example of static model.

Dynamic model
- It can be viewed as a collection of procedures or behaviors that taken together reflect the behavior of a system over time. Dynamic modeling is the most useful during the design and implementation phases of the system development. The UML interaction diagrams and activity models are examples of dynamic models.

WHY MODELING?

- Clarity
- Familiarity
- Maintenance
- Simplification

Advantages of Modeling

Good models are essential for communication among project teams. As the complexity of systems increases, so does the importance of good modeling techniques. Some of the advantages are as follows:
- Models make it easier to express complex ideas.
- The main reason for modeling is to reduction of complexity.
- Models enhance and reinforce learning and training.
- The cost of modeling analysis is much lower than the cost of similar perimentation conducted in real system.
- Manipulation of the model is much easier.
INTRODUCTION TO UNIFIED MODELING LANGUAGE:

The unified modeling language (UML) is a language for specifying constructing, visualizing and documenting the software system and its components. The UML is a graphical language with sets of rules and semantics. The primary goals in the design of the UML were as follows:

◊ Provide users a ready-to-use, expressive visual modeling language so they can develop and exchange meaningful models.
◊ Provide extensibility and specialization mechanisms to extend the core concepts.
◊ Be independent of particular programming languages and development processes.
◊ Provide a formal basis for understanding the modeling language.
◊ Encourage the growth of the OO tools market.
◊ Support higher-level development concepts.
◊ Integrate best practices and methodologies.

Nine UML graphical diagrams:
 a. Class diagram(static)
 b. Use-case diagram
 c. Behavior diagram(dynamic)
   i. Interaction diagram
      1. Sequence diagram
      2. Collaboration diagram
   ii. State chart diagram
   iii. Activity diagram
 d. Implementation diagram.
   i. Component diagram
   ii. Deployment diagram.
 e. Object diagram
What is UML?
UML stands for Unified Modeling Language. This object-oriented system of notation has evolved from the work of Grady Booch, James Rumbaugh, Ivar Jacobson, and the Rational Software Corporation. These renowned computer scientists fused their respective technologies into a single, standardized model. Today, UML is accepted by the Object Management Group (OMG) as the standard for modeling object oriented programs.

Types of UML Diagrams

Class diagrams
Class diagrams are the backbone of almost every object oriented method, including UML. They describe the static structure of a system.

Package diagrams
Package diagrams are a subset of class diagrams, but developers sometimes treat them as a separate technique. Package diagrams organize elements of a system into related groups to minimize dependencies between packages.

Object diagrams
Object diagrams describe the static structure of a system at a particular time. They can be used to test class diagrams for accuracy.

Use case diagrams
Use case diagrams model the functionality of system using actors and use cases.

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

Collaboration diagrams
Collaboration diagrams represent interactions between objects as a series of sequenced messages. Collaboration diagrams describe both the static structure and the dynamic behavior of a system.

Statechart diagrams
Statechart diagrams describe the dynamic behavior of a system in response to external stimuli. Statechart diagrams are especially useful in modeling reactive objects whose states are triggered by specific events.
Activity diagrams
Activity diagrams illustrate 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. Typically, activity diagrams are used to model workflow or business processes and internal operation.

Component diagrams
Component diagrams describe the organization of physical software components, including source code, run-time (binary) code, and executables.

Deployment diagrams
Deployment diagrams depict the physical resources in a system, including nodes, components, and connections.

Types of UML Diagrams

Each UML diagram is designed to let developers and customers view a software system from a different perspective and in varying degrees of abstraction. UML diagrams commonly created in visual modeling tools include:  

Use Case Diagram displays the relationship among actors and use cases.  

Class Diagram models class structure and contents using design elements such as classes, packages and objects. It also displays relationships such as containment, inheritance, associations and others.

Interaction Diagrams

- Sequence Diagram displays the time sequence of the objects participating in the interaction. This consists of the vertical dimension (time) and horizontal dimension (different objects).
- Collaboration Diagram displays an interaction organized around the objects and their links to one another. Numbers are used to show the sequence of messages.

State Diagram displays the sequences of states that an object of an interaction goes through during its life in response to received stimuli, together with its responses and actions.
**Activity Diagram** displays a special state diagram where most of the states are action states and most of the transitions are triggered by completion of the actions in the source states. This diagram focuses on flows driven by internal processing.  

**Physical Diagrams**

- **Component Diagram** displays the high level packaged structure of the code itself. Dependencies among components are shown, including source code components, binary code components, and executable components. Some components exist at compile time, at link time, at run times well as at more than one time.  
- **Deployment Diagram** displays the configuration of run-time processing elements and the software components, processes, and objects that live on them. Software component instances represent run-time manifestations of code units.  

**Introduction**

- UML represents a unification of the concepts and notations of Booch, Rumbaugh, and Jacobson.  
- Notation plays an important role in modeling and the goal of the UML is to become a universal notation for creating models of object-oriented (OO) software.  
- An object-oriented method comprises both a process (e.g., Objectory, Fusion) and a modeling notation (e.g., Booch, OMT, UML). The Unified Modeling Language (UML) is a modeling notation and does not prescribe a specific process.  
- UML is language-independent and vendor-independent. For example, UML can be used in Java, Smalltalk, or C++ development efforts. Further, the UML can be used in conjunction with tools such as Visio, Rational Rose or TogetherJ.  
- UML is an extensible language. For example, stereotypes are one of the mechanisms that can be used to extend the UML. In general, a stereotype represents a usage distinction. Stereotypes can be used with any diagram to extend its meaning.
UML Notation

There are many different types of UML notation. For this short presentation, a few types of notation will be covered. For a more complete description of the UML notation, you can refer to the UML 1.1 Notation Guide. When beginning to use the UML, it is helpful to start with some basic notation (e.g., class diagrams) and, then, as you become more proficient, advanced UML notation can be used.

Class Diagram
Class diagrams represent the static structure of the classes and their relationships (e.g., inheritance, aggregation) in a system. A class icon is divided into three components?class name, attributes, and operations.

- The access modifier symbols +, -, and # correspond respectively to public, private, and protected access modifiers.
- Realizes relationship from a class to an interface indicates that the class implements the operations specified in the interface.
- Associations represent relationships between instances of classes. Each association has two roles; each role is a direction on the association.
- A role has multiplicity (an indication of how many objects may participate in a given relationship). Examples of multiplicity indicators are: 1 (Exactly one), * (0 to any positive integer), 1..* (1 to any positive integer), 0..1 (0 or 1).
- Constraint specifies conditions among model elements that must be maintained as true. Constraints are shown using text enclosed in braces.
- Generalization relationship is an association with a small triangle next to the class being inherited from whereas an aggregation relationship is an association with a diamond next to the class representing the aggregate.
Class diagrams can be used to document a Business Object Model and are widely used in both high-level and low-level design documents.

From a Java perspective, it may not be effective to include each and every operation and attribute of a class in a class diagram because this leads to maintenance issues (e.g., if a method or attribute changes in the code then the model needs to be updated); rather, javadoc can be used to document an implementation.

To maintain consistency with Java coding style, class names should begin with an uppercase letter while operations and attributes should begin with a lowercase letter.
- Generalization relationships can be mapped to the Java extends relationship while the realizes relationship can be mapped to the Java implements relationship.

**Class diagrams** are widely used to describe the types of objects in a system and their relationships. Class diagrams model class structure and contents using design elements such as classes, packages and objects. Class diagrams describe three different perspectives when designing a system, conceptual, specification, and implementation. These perspectives become evident as the diagram is created and help solidify the design. This example is only meant as an introduction to the UML and class diagrams. If you would like to learn more see the Resources page for more detailed resources on UML.

Classes are composed of three things: a name, attributes, and operations. Below is an example of a class.

![Class Diagram Example](image)

Class diagrams also display relationships such as containment, inheritance, associations and others. Below is an example of an associative relationship:

![Association Diagram Example](image)
The association relationship is the most common relationship in a class diagram. The association shows the relationship between instances of classes. For example, the class Order is associated with the class Customer. The multiplicity of the association denotes the number of objects that can participate in the relationship. For example, an Order object can be associated to only one customer, but a customer can be associated to many orders.

Another common relationship in class diagrams is a generalization. A generalization is used when two classes are similar, but have some differences. Look at the generalization below:

In this example the classes Corporate Customer and Personal Customer have some similarities such as name and address, but each class has some of its own attributes and operations. The class Customer is a general form of both the Corporate Customer and Personal Customer classes. This allows the designers to just use the Customer class for modules and do not require in-depth representation of each type of customer.
When to Use: Class Diagrams

Class diagrams are used in nearly all Object Oriented software designs. Use them to describe the Classes of the system and their relationships to each other.

How to Draw: Class Diagrams

Class diagrams are some of the most difficult UML diagrams to draw. To draw detailed and useful diagrams a person would have to study UML and Object Oriented principles for a long time. Therefore, this page will give a very high level overview of the process. To find list of where to find more information see the Resources page.

Before drawing a class diagram consider the three different perspectives of the system the diagram will present; conceptual, specification, and implementation. Try not to focus on one perspective and try see how they all work together.

When designing classes consider what attributes and operations it will have. Then try to determine how instances of the classes will interact with each other. These are the very first steps of many in developing a class diagram. However, using just these basic techniques one can develop a complete view of the software system.
This example is only meant as an introduction to the UML and use cases. If you would like to learn more see the Resources page for more detailed resources on UML.

**Use case diagram:**

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.
An actor is represents a user or another system that will interact with the system you are modeling. A use case is an external view of the system that represents some action the user might perform in order to complete a task.

**When to Use: Use Cases Diagrams**

Use cases are used in almost every project. They 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.

**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. If you would like to learn more see the Resources page for more detailed resources on UML.

Start by listing a sequence of steps a user might take in order to complete an action. For example a user placing an order with a sales company might follow these steps.

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

These steps would generate this simple use case diagram:
This example shows the customer as a 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.

**Package Diagram**

Package diagrams provide a mechanism for dividing and grouping model elements
(e.g., classes, use cases). In UML, a package is represented as a folder.

- In effect, a package provides a namespace such that two different elements
  in two different packages can have the same name.
- Packages may be nested within other packages.
- Dependencies between two packages reflect dependencies between any two
classes in the packages. For example, if a class in Package A uses the
services of a class in Package B, Package A is dependent on Package B. An
important design consideration is the minimization of dependencies between
packages.
Source: Fowler & Scott, UML Distilled

- Package diagrams can be used in a high-level design or architecture document to describe a system’s overall structure, can support project planning and also be used as a unit to perform testing.
- A package can be considered logically equivalent to a Java package and dependencies between packages can be mapped to Java import statements.

Use Case Diagram
Use cases are used to obtain system requirements from a user's perspective. A use case can be described as a interaction that a user has with a system to achieve a goal. It is helpful to provide a template that can be used to document use cases. Note that a use case template is not part of the UML notation. A typical use case can be documented using the following template:
UML use case diagram syntax

- Actor – participant in use case; usually a user of the system, may not be a human.

An individual use case.

- Communication association.

- Generalisation relationship.

- Dependency relationship.

Use Case: Create an email with no attachment
Reference: 1
Author: John Doe
Pre-condition: User has logged into a system (the necessary conditions that have to be met before the use case can be performed)
Description: User enters recipient address, subject and text message. Then, user selects to send message. (description of the basic course of interactions between the actor and the system. If prose format is selected, a general form for describing the steps in a use case is: <time or sequence factor> ? <actor> ? <action> ? <constraints>.)
Post-condition: Mail message is sent (the state of the system after the use case is
Exceptions: Recipient address not entered (different error situations that can occur)
Variations: Create an email with an attachment (alternative courses that can potentially be taken)
Related use cases: Login use case
Non-functional requirements (Priority, Response Time, etc.): High priority

A use case diagram provides an overview of the system's use cases and their actors; it places use cases in context. In UML, an actor is represented as a stickman while a use case is represented as an oval.

- An actor is a role that a user plays with respect to the system.
- Two types of relationships are used in use case diagrams.
- The extends relationship is used when one use case is similar to another use case but does a bit more or to describe optional behavior (e.g., forward a mail message).
- The uses relationship occurs when you have common behavior that exists in multiple use cases that can be factored out into a separate use case (e.g., login use case). Note that actors have relationships to use cases that are being extended whereas there is often no actor associated with a common use case.
Use case diagrams can be used in product specifications, software architecture documents, and system test documents to provide an overview of the system requirements.

Use cases can specify prioritization that can support a phased delivery of a system.

Use cases can also be used to develop test plans and user guides.

Source: Fowler & Scott, UML Distilled
Interaction Diagrams: Sequence and Collaboration

Interaction diagrams show how objects interact with one another. There are two types of interaction diagrams: Collaboration Diagram and Sequence Diagram.

Collaboration diagrams can be used to show how objects in a system interact over multiple use cases. Collaboration diagrams are helpful during the exploratory phases of a development process (i.e., trying to search for objects and their relationships). Since there is no explicit representation of time in Collaboration Diagrams, the messages are numbered to denote the sending order.

Interaction diagrams can be used during the high-level design (e.g., for business object model development) as well as for low-level design activities (e.g., developing an implementation for a use case). Interaction diagrams are helpful for showing the collaborations between objects but fall short in terms of a complete definition of the behavior.

Interaction diagrams model the behavior of use cases by describing the way groups of objects interact to complete the task. The two kinds of interaction diagrams are sequence and collaboration diagrams. This example is only meant as an introduction to the UML and interaction diagrams. If you would like to learn more see the Resources page for a list of more detailed resources on UML.

When to Use: Interaction Diagrams

Interaction diagrams are used when you want to model the behavior of several objects in a use case. They demonstrate how the objects collaborate for the behavior. Interaction diagrams do not give a in depth representation of the behavior. If you want to see what a specific object is doing for several use cases use a state diagram. To see a particular behavior over many use cases or threads use an activity diagrams. ¹

How to Draw: Interaction Diagrams

Sequence diagrams, collaboration diagrams, or both diagrams can be used to demonstrate the interaction of objects in a use case. Sequence diagrams generally show the sequence of events that occur. Collaboration diagrams demonstrate how objects are statically connected. Both diagrams are relatively simple to draw and contain similar elements. ¹
Sequence diagrams:

Sequence diagrams demonstrate the behavior of objects in a use case by describing the objects and the messages they pass. The diagrams are read left to right and descending. The example below shows an object of class 1 start the behavior by sending a message to an object of class 2. Messages pass between the different objects until the object of class 1 receives the final message.

Below is a slightly more complex example. The light blue vertical rectangles the objects activation while the green vertical dashed lines represent the life of the object. The green vertical rectangles represent when a particular object has control. The ■ represents when the object is destroyed. This diagrams also shows conditions for messages to be sent to other object. The condition is listed between brackets next to the message. For example, a [condition] has to be met before the object of class 2 can send a message() to the object of class 3.
The next diagram shows the beginning of a sequence diagram for placing an order. The object an Order Entry Window is created and sends a message to an Order object to prepare the order. Notice the the names of the objects are followed by a colon. The names of the classes the objects belong to do not have to be listed. However the colon is required to denote that it is the name of an object following the objectName:className naming system.

Next the Order object checks to see if the item is in stock and if the [InStock] condition is met it sends a message to create an new Delivery Item object.
The next diagrams adds another conditional message to the Order object. If the item is [OutOfStock] it sends a message back to the Order Entry Window object stating that the object is out of stack.
This simple diagram shows the sequence that messages are passed between objects to complete a use case for ordering an item.

**Collaboration diagrams:**

Collaboration diagrams are also relatively easy to draw. They show the relationship between objects and the order of messages passed between them. The objects are listed as icons and arrows indicate the messages being passed between them. The numbers next to the messages are called sequence numbers. As the name suggests, they show the sequence of the messages as they are passed between the objects. There are many acceptable sequence numbering schemes in UML. A simple 1, 2, 3... format can be used, as the example below shows, or for more detailed and complex diagrams a 1, 1.1 ,1.2, 1.2.1... scheme can be used.
The example below shows a simple collaboration diagram for the placing an order use case. This time the names of the objects appear after the colon, such as :Order Entry Window following the objectName:className naming convention. This time the class name is shown to demonstrate that all of objects of that class will behave the same way.
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 track the different states of its objects through the system.

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 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. State diagrams are other combined with other diagrams such as interaction diagrams and activity diagrams.¹

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 indicating 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.

All state diagrams begin with an initial state of the object. 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.
Below is an example of a state diagram might look like for an Order object. 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.
State diagrams can also show a super-state for the object. A super-state is used when many transitions lead to the a certain state. Instead of showing all of the transitions from each state to the redundant state a super-state can be used to show that all of the states inside of the super-state can transition to the redundant state. This helps make the state diagram easier to read.

The diagram below shows a super-state. Both the Checking and Dispatching states can transition into the Canceled state, so a transition is shown from a super-state 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.
STATE CHART DIAGRAM FOR ATM system

State-Chart for One Session

READING CARD

Card read successfully

READING PIN

PIN read successfully

CHOOSING TRANSACTION

transaction chosen

PERFORMING TRANSACTIONS

include Transaction

aborted due to too many invalid PINS - card retained

Customer finished

EJECTING CARD

display "Card not readable"

Card not readable
Activity Diagrams
Activity diagrams show behavior with control structure. Activity diagrams can be used to show behaviors over many use case, model business workflows, or describe complicated methods.

- Activities in a diagram may or may not correspond to methods.
- Specific notation found in this type of diagram includes guards which are logical expressions that evaluate to true or false.
- A synchronization bar indicates that the outbound trigger occurs only after all inbound triggers have occurred.
- Swimlanes (using a swimming pool analogy) allow you to vertically partition an activity diagram so that the activities in each lane represent the responsibilities of a particular class or department.
- Activity diagrams are helpful early in the modeling of a process to help you understand the overall process. Useful when you want to show parallel processes or for multithreaded programs.
- Disadvantage of activity diagrams is that they do not make explicit which objects execute which activities, and the way that the messaging works between them.

Activity diagrams describe the workflow behavior of a system. Activity diagrams are similar to state diagrams because activities are the state of doing something. The diagrams describe the state of activities by showing the sequence of activities performed. Activity diagrams can show activities that are conditional or parallel.

**When to Use: Activity Diagrams**

Activity diagrams should be used in conjunction with other modeling techniques such as interaction diagrams and state diagrams. The main reason to use activity diagrams is to model the workflow behind the system being designed. Activity Diagrams are also useful for: analyzing a use case by describing what actions need to take place and when they should occur; describing a complicated sequential algorithm; and modeling applications with parallel processes.

However, activity diagrams should not take the place of interaction diagrams and state diagrams. Activity diagrams do not give detail about how objects behave or how objects collaborate.

**How to Draw: Activity Diagrams**

Activity diagrams show the flow of activities through the system. Diagrams are read from top to bottom and have branches and forks to describe conditions and parallel activities. A fork is used when multiple activities are occurring at the same time. The diagram below shows a fork after activity1. This indicates that both activity2 and activity3 are occurring at the same time. After activity2 there is a branch. The branch describes what activities will take place based on a set of conditions. All branches at some point are followed by a merge to indicate the end of the conditional behavior started by that branch. After the merge all of the
parallel activities must be combined by a join before transitioning into the final activity state.

**Swimlane diagram**

Below is a possible activity diagram for processing an order. The diagram shows the flow of actions in the system's workflow. Once the order is received the activities split into two parallel sets of activities. One side fills and sends the order while the other handles the billing. On the Fill Order side, the method of delivery is decided conditionally. Depending on the condition either the Overnight
Delivery activity or the Regular Delivery activity is performed. Finally the parallel activities combine to close the order.
Types of physical diagrams

There are two types of physical diagrams: **deployment diagrams** and **component diagrams**. Deployment diagrams show the physical relationship between hardware and software in a system. Component diagrams show the software components of a system and how they are related to each other. These relationships are called dependencies.  

**When to Use: Physical Diagrams**

Physical diagrams are used when development of the system is complete. Physical diagrams are used to give descriptions of the physical information about a system.

**How to Draw: Physical Diagrams**

Many times the deployment and component diagrams are combined into one physical diagram. A combined deployment and component diagram combines the features of both diagrams into one diagram.

The deployment diagram contains nodes and connections. A node usually represents a piece of hardware in the system. A connection depicts the communication path used by the hardware to communicate and usually indicates a method such as TCP/IP.
The component diagram contains components and dependencies. Components represent the physical packaging of a module of code. The dependencies between the components show how changes made to one component may affect the other components in the system. Dependencies in a component diagram are represented by a dashed line between two or more components. Component diagrams can also show the interfaces used by the components to communicate to each other.  

The combined deployment and component diagram below gives a high level physical description of the completed system. The diagram shows two nodes which represent two machines communicating through TCP/IP. Component2 is dependant on component1, so changes to component 2 could affect component1. The diagram also depicts component3 interfacing with component1. This diagram gives the reader a quick overall view of the entire system.
Patterns

A pattern is instructive information that captures the essential structure and insight of a successful family of proven solutions to a recurring problem that arises within a certain context and system of forces.

- The main idea behind using patterns is to provide documentation to help categorize and communicate about solutions to recurring problems.

- The pattern has a name to facilitate discussion and the information it represents.

A good pattern will do the following:
- It solves a problem.
- Patterns capture solutions, not just abstract principles or strategies.
➢ It is a proven concept.
➢ Patterns capture solutions with a track record, not theories or speculation.
➢ The solution is not obvious.
➢ The best patterns generate a solution to a problem indirectly—a necessary approach for the most difficult problems of design.
➢ It describes a relationship.
➢ Patterns do not just describe modules, but describe deeper system structures and mechanisms.
➢ The pattern has a significant human component.
➢ All software serves human comfort or quality of life; the best patterns explicitly appeal to aesthetics and utility.

**Generative Patterns:**
Patterns that suggest the way of finding the solution

**Non Generative patterns:**
They do not suggest instead they give a passive solution. Non Generative patterns cannot be used in the entire situation.

**Patterns template**
There are different pattern templates are available which will represent a pattern. It is generally agreed that a pattern should contain certain following components

<table>
<thead>
<tr>
<th><strong>Name</strong></th>
<th>A meaningful name.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Problem</strong></td>
<td>A statement of the problem that describes its intent.</td>
</tr>
<tr>
<td><strong>Context</strong></td>
<td>The preconditions under which the problem and its solution seem to recur and for which the solution is desirable. This tells us the pattern’s applicability.</td>
</tr>
<tr>
<td><strong>Forces</strong></td>
<td>constraints and conflicts with one another with the goals which we wish to achieve.</td>
</tr>
<tr>
<td><strong>Solution</strong></td>
<td>solution makes the pattern come alive.</td>
</tr>
<tr>
<td><strong>Examples</strong></td>
<td>sample implementation</td>
</tr>
<tr>
<td><strong>Resulting context</strong></td>
<td>describes the post conditions and side effects of the pattern.</td>
</tr>
<tr>
<td><strong>Rationale</strong></td>
<td>justifying explanation of steps or rules in the pattern. This tells how the pattern actually works, why it works and why it is good.</td>
</tr>
<tr>
<td><strong>Related patterns.</strong></td>
<td>The static and dynamic relationships between these patterns and others with in the same pattern language or system.</td>
</tr>
</tbody>
</table>
**Known uses**  The known occurrences of the pattern and its application within existing systems.

**Anti patterns**
A pattern represents a best practice whereas an anti pattern represents worst practice or a lesson learned.

**Anti patterns come in two varieties:**
- Those describing a bad solution to a problem that resulted in a bad situation
- Those describing how to get out of a bad situation and how to proceed from there to a good solution.

**Capturing Patterns**
- Guidelines for capturing patterns:
  - Focus on practicability.
  - Aggressive disregard of originality.
  - Nonanonymouse review.
  - Writers' workshops instead of presentations.
  - Careful editing.

**Frameworks:**
- Frameworks are the way of delivering application development patterns to support/share best development practice during application development.

  - In general framework is a generic solution to a problem that can be applied to all levels of development. Design and Software frameworks and most popular where Design pattern helps on Design phase and software frameworks help in Component Based Development phase.

  - Framework groups a set of classes which are either concrete or abstract. This group can be sub classed in to a particular application and recomposing the necessary items.

  - Frameworks can be inserted in to a code where a design pattern cannot be inserted. To include a design pattern the implementation of the design pattern is used.

  - Design patterns are instructive information; hence they are less in space where Frameworks are large in size because they contain many design patterns.
Frameworks are more particular about the application domain where design patterns are less specified about the application domain.

**Differences Between Design Patterns and Frameworks**
- Design patterns are more abstract than frameworks.
- Design patterns are smaller architectural elements than frameworks.
- Design patterns are less specialized than frameworks.

**Conclusion**

In this overview, the UML was discussed in terms of Class Diagrams, Package Diagrams, Interaction Diagrams, Use Case Diagrams, and Activity Diagrams. Other types of UML notation that were not discussed include State Diagrams (explore the state transitions of a single object), Deployment Diagrams (show how software is mapped onto hardware devices), and Component Diagrams (show the structure of software in terms of software components and their relationship to executable components). Information on these and other types of UML notation can be found in the references at the end of this presentation.

**References**

**Modeling Tools**
Rational Rose http://www.rational.com/
TogetherJ http://www.togetherj.com/

**Web Sites**
http://www.holub.com/ UML Quick Reference guide
http://www.ambysoft.com/ Whitepapers on a variety of UML-related topics
www.softdocwiz.com/UML.html UML Dictionary which is a supplement to Fowler's UML Distilled book

**Books**
Muller, Instant UML
Fowler and Scott, UML Distilled
Eriksson and Pensker, UML Toolkit
Booch, Rumbaugh, and Jacobson, The Unified Modeling Language User Guide
UML Notation Guide, Version 1.1
Coad, North, and Mayfield, Object Models: Strategies, Patterns, and Applications
SUMMARY

A model is an abstract representation of a system, constructed to understand the system prior to building or modifying it. It is a model of a simplified representation of reality. Models can represent static or dynamic situations. The unified modeling language (UML) is a language for specifying constructing, visualizing and documenting the software system and its components. The UML is a graphical language with sets of rules and semantics. The primary goals in the design of the UML were as follows:

◊ Provide users a ready-to-use, expressive visual modeling language so they can develop and exchange meaningful models
◊ Provide extensibility and specialization mechanisms to extend the core concepts.
◊ Be independent of particular programming languages and development processes.
◊ Provide a formal basis for understanding the modeling language.
◊ Encourage the growth of the OO tools market.
◊ Support higher-level development concepts.
◊ Integrate best practices and methodologies.

KEY TERMS

Model
Static model
Dynamic model
UML Class diagrams
UML Use case diagrams
UML Sequence diagrams
UML Collaboration diagrams
UML State chart diagrams
UML Activity diagrams
UML Component diagrams
UML Deployment diagrams
KEY TERM QUIZ

1. A **model** is an abstract representation of a system, constructed to understand the system prior to building or modifying it.
2. **Static model** can be viewed as a snapshot of a system’s parameters at rest or a specific point in time.
3. **Dynamic model** can be viewed as a collection of procedures or behaviors that take together reflect the behavior of a system over time.
4. The **unified modeling language (UML)** is a language for specifying constructing, visualizing and documenting the software system and its components.
5. **Class diagrams** are the backbone of almost every object oriented method, including UML. They describe the static structure of a system.
6. **Package diagrams** are a subset of class diagrams, but developers sometimes treat them as a separate technique. Package diagrams organize elements of a system into related groups to minimize dependencies between packages.
7. **Object diagrams** describe the static structure of a system at a particular time. They can be used to test class diagrams for accuracy.
8. **Use case diagrams** model the functionality of system using actors and use cases.
9. **Sequence diagrams** describe interactions among classes in terms of an exchange of messages over time.
10. **Collaboration diagrams** represent interactions between objects as a series of sequenced messages. Collaboration diagrams describe both the static structure and the dynamic behavior of a system.
11. **Statechart diagrams** describe the dynamic behavior of a system in response to external stimuli.
12. **Statechart diagrams** are especially useful in modeling reactive objects whose states are triggered by specific events.
13. **Activity diagrams** illustrate 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. Typically, activity diagrams are used to model workflow or business processes and internal operation.
14. **Component diagrams** describe the organization of physical software components, including source code, run-time (binary) code, and executables.
15. **Deployment diagrams** depict the physical resources in a system, including nodes, components, and connections.
MULTIPLE CHOICE

1. A model is an abstract representation of a system, constructed to understand the system prior to building or modifying it.
   (a) model (b) static model (c) dynamic model
2. Static model can be viewed as a snapshot of a system’s parameters at rest or a specific point in time.
   (a) model (b) static model (c) dynamic model
3. Dynamic model can be viewed as a collection of procedures or behaviors that take together reflect the behavior of a system over time.
   (a) model (b) static model (c) dynamic model
4. The unified modeling language (UML) is a language for specifying constructing, visualizing and documenting the software system and its components.
   (a) UML (b) static model (c) dynamic model
5. Class diagrams are the backbone of almost every object oriented method, including UML. They describe the static structure of a system.
   (a) Class diagrams (b) Package diagrams (c) Object diagrams
6. Package diagrams are a subset of class diagrams, but developers sometimes treat them as a separate technique. Package diagrams organize elements of a system into related groups to minimize dependencies between packages.
   a) Class diagrams (b) Package diagrams (c) Object diagrams
7. Object diagrams describe the static structure of a system at a particular time. They can be used to test class diagrams for accuracy.
   a) Class diagrams (b) Package diagrams (c) Object diagrams
8. Use case diagrams model the functionality of system using actors and use cases.
   a) use case diagrams (b) collaboration diagrams (c) sequence diagrams
9. Sequence diagrams describe interactions among classes in terms of an exchange of messages over time.
   a) use case diagrams (b) collaboration diagrams (c) sequence diagrams
10. Collaboration diagrams represent interactions between objects as a series of sequenced messages.
    a) use case diagrams (b) collaboration diagrams (c) sequence diagrams
11. Statechart diagrams describe the dynamic behavior of a system in response to external stimuli.
    a) state chart diagrams (b) activity diagrams (c) component diagrams
12. Statechart diagrams are especially useful in modeling reactive objects whose states are triggered by specific events.
    a) state chart diagrams (b) activity diagrams (c) component diagrams
13. **Activity diagrams** illustrate 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. Typically, activity diagrams are used to model workflow or business processes and internal operation.
   a) state chart diagrams (b) activity diagrams (c) component diagrams

14. **Component diagrams** describe the organization of physical software components, including source code, run-time (binary) code, and executables.
   a) state chart diagrams (b) activity diagrams (c) component diagrams

15. **Deployment diagrams** depict the physical resources in a system, including nodes, components, and connections.
   a) state chart diagrams (b) deployment diagrams (c) component diagrams

**REVIEW QUESTIONS:**

1. What is a model? (Pg 89)
2. Why do we need to model a problem? (Pg 90)
3. What are the different types of modeling? Briefly describe each
   a. (Pg 90-91)
4. What is UML? What is the importance of UML? (Pg 92-93)
5. Describe the class diagram (Pg 94)
6. How would you represent or model externally visible behavior of a class? (Pg 95)
7. What is an association role? (Pg 95)
8. What is multiplicity? (Pg 97)
9. What is a qualifier? (Pg 96)
10. What are some of the forms of associations? Draw their UML representations (Pg 97-99)
11. How would you show complete and incomplete generalizations?
    a. (Pg 99-101)
12. What are model constraints and how are they represented in the UML? (Pg 116)
13. Name and describe the relationships in a use case diagram (Pg 103)
14. What are some of the UML dynamic diagrams? (Pg 103)
15. When would you use interaction diagrams? (Pg 104)
16. Differentiate between sequence and collaboration diagrams
    a. (Pg 104-106)
17. What is the purpose of an activity model? (Pg 109)
18. What is a meta-model? Is understanding a meta-model important?
    a. (Pg 117-118)
19. Why do we go in for modeling? (Pg 91)
20. Give the benefits of modeling (Pg 91)
21. List out the advantages of modeling (Pg 92)
22. List out the goals in the design of UML (Pg 93)
23. Name the UML diagrams (Pg 93)
24. What is the significance of sequence diagram? (Pg 106)
25. What is a swimlane? (Pg 111)
26. What are implementation diagrams? (Pg 111)
27. Differentiate between component and deployment diagrams
   a. (Pg 112-113)
28. What is a package? (Pg 114)
29. What is model dependency? (Pg 115)
30. What is a note? (Pg 117)
31. What is a stereotype? (Pg 117)

Reference: