# THE COPPERBELT UNIVERSITY

COMPUTER SCIENCE DEPARTMENT

## CS 432 – OBJECT ORIENTED SOFTWARE ENGINEERING

COMPILED BY DR NTALASHA

### Background and Rationale

This course separates and makes explicit the decisions that make up an OO analysis and design. It shows how to use the UML notation most effectively. The rationale of this course is to make the student understand in broad terms the concepts of object oriented systems analysis and design. They should be able to describe the work of the system analysts.

### Course aim

This course separates and makes explicit the decisions that make up an OO analysis and design. We show how to use the UML notation most effectively both to discuss designs with colleagues and in documents. UML is now an industry standard supported by the entire major tools. The course provides especially strong coherence between the different notations used in UML, making it clear when there are inconsistencies or holes in the analysis.

### Learning Outcomes

At the end of the course students will be expected to;

- Use the knowledge obtained to analyze the systems;
- Design a new system and offer recommendations

### Course contents

**Understanding the Basics: Object oriented concepts**

**Object-Oriented Programming**:
Introducing the principles of OO design, aims of OO design, OO designs are based on the "real" world, what is an object? Distribution of responsibilities, objects, classes, messages and methods, decoupling for flexible software, interfaces and the "implements" relation, types and object specifications, class extension.

**Object-Oriented Analysis And Design**:
Aims and overview of process, layering design decisions: abstraction, exposing gaps and inconsistencies: precision, traceable designs: continuity; clear, ready **communication**: a language for design, a vanilla process: development from scratch (brief overview), business modelling: concepts and tasks, systems requirements models, responsibilities and collaborations, persistence, GUI, distribution, coding in an OO language, component-based design (brief overview), robust flexible software, components and interfaces, component kits and architecture, component and re-use culture, patterns (brief overview)

**Business Modeling And UML Basics**:
This section covers techniques of identifying business concepts and tasks, and introduces relevant parts of UML along the way. Static models, objects, types, attributes, snapshot, subtypes, dynamics, use cases and tasks, event charts, state charts, building a business model, finding a business model, finding use-cases, connecting use-case and class views, the dictionary, UML notation review, uses of business models, architecture of business process, context of software requirements, basis for component interface definition, documentation style

**Requirements Modeling**: This section deals with the specification of requirements of a software component, application, or complete system. More modeling patterns and techniques are investigated. System context models, high-level operation specifications, state charts for system models, meaning of “model”, how to start abstract and get more detailed, event charts: horizontal and vertical expansion, elaborating models, relating the levels of detail, building a system specification, system context, defining use-case goals, modelling patterns

**Basic Design**: The key principle of OO design is assigning responsibilities and designing collaborations. Separating core from GUI, persistence and other layers, selection of control objects, designing system operations with messages, decoupling, extensibility, reusability, CRC ”cards”, dependencies and visibilities, the class dictionary, translation to code (Java, C++ or Smalltalk examples).

**Prescribed Textbooks**

1. Schach, Stephen R., Introduction to Object-Oriented Analysis and Design, McGraw Hill 2003

**Recommended Text**

1. Object-Oriented and Classical Software Engineering. Schach, Stephen R. McGraw Hill 7th Edition 2006
2. Object-Oriented Design and Patterns. Horstmann, Cay S. John Wiley and Sons2nd Edition 2005
3. Object-Oriented System Development: A Gentle Introduction. Britton, Carol; Doake, Jill. McGraw Hill 2000

# THE COPPERBELT UNIVERSITY

COMPUTER SCIENCE DEPARTMENT

## CS 432 – OBJECT ORIENTED SYSTEMS ANALYSIS AND DESIGN

COMPILED BY DR NTALASHA

## UNIT ONE –

### A Brief History

The object-oriented paradigm took its shape from the initial concept of a new programming approach, while the interest in design and analysis methods came much later.

- The first object–oriented language was Simula (Simulation of real systems) that was developed in 1960 by researchers at the Norwegian Computing Center.
- In 1970, Alan Kay and his research group at Xerox PARK created a personal computer named Dynabook and the first pure object-oriented programming language (OOPL)-Smalltalk, for programming the Dynabook.
- In the 1980s, Grady Booch published a paper titled Object Oriented Design that mainly presented a design for the programming language, Ada. In the ensuing editions, he extended his ideas to a complete object– oriented design method.

In the 1990s, Coad incorporated behavioral ideas to object-oriented methods.

The other significant innovations were Object Modelling Techniques (OMT) by James Rumbaugh and Object-Oriented Software Engineering (OOSE) by Ivar Jacobson.

### Object-Oriented Analysis

Object–Oriented Analysis (OOA) is the procedure of identifying software engineering requirements and developing software specifications in terms of a software system's object model, which comprises of interacting objects.

The main difference between object-oriented analysis and other forms of analysis is that in object-oriented approach, requirements are organized around objects, which integrate both data and functions. They are modelled after real-world objects that the system interacts with. In traditional analysis methodologies, the two aspects - functions and data - are considered separately.

Grady Booch has defined OOA as, *"Object-oriented analysis is a method of analysis that examines requirements from the perspective of the classes and objects found in the vocabulary of the problem domain"*.

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

- Identifying objects
- Organizing the objects by creating object model diagram
- Defining the internals of the objects, or object attributes
- Defining the behavior of the objects, i.e., object actions
- Describing how the objects interact

The common models used in OOA are use cases and object models.

### Object-Oriented Design

Object–Oriented Design (OOD) involves implementation of the conceptual model produced during object-oriented analysis. In OOD, concepts in the analysis model, which are technology independent, are mapped onto implementing classes, constraints are identified and interfaces are designed, resulting in a model for the solution domain, i.e., a detailed description of how the system is to be built on concrete technologies. The implementation details generally include:
Restructuring the class data (if necessary),

- Implementation of methods, i.e., internal data structures and algorithms,
- Implementation of control, and
- Implementation of associations.

Grady Booch has defined object-oriented design as “a method of design encompassing the process of object-oriented decomposition and a notation for depicting both logical and physical as well as static and dynamic models of the system under design”.

## 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 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, one such model is Unified Modeling Language (UML).

THE OBJECT MODEL

Object oriented development offers a different model from the traditional software development approach, which is based on functions and procedures.

- An Object-Oriented environment, software is a collection of discrete objects that encapsulate their data and the functionality to model real world “Objects”.
- Object are defined, it will perform their desired functions and seal them off in our mind like black boxes.
- The object- Oriented life cycle encourages a view of the world as a system of cooperative and collaborating agents.
- An objective orientation producers system that are easier evolve, move flexible more robust, and more reusable than a top-down structure approach.
- An object orientation allows working at a higher level of abstraction.
- It provides a seamless transition among different phases of software development.
- It encourages good development practices.
- It promotes reusability.

The unified Approach (UA) is the methodology for software development proposed and used and the following concepts consist of Unified Approach

## Object-Oriented Programming

Object-oriented programming (OOP) is a programming paradigm based upon objects (having both data and methods) that aims to incorporate the advantages of modularity and reusability. Objects, which are usually instances of classes, are used to interact with one another to design applications and computer programs.

***The important features of object–oriented programming are:***

- Bottom up approach in program design
- Programs organized around objects, grouped in classes
- Focus on data with methods to operate upon object’s data
- Interaction between objects through functions
- Reusability of design through creation of new classes by adding features to existing classes

Some examples of object-oriented programming languages are C++, Java, Smalltalk, Delphi, C#, Perl, Python, Ruby, and PHP.

Grady Booch has defined object–oriented programming as “a method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships”.

## OBJECT MODEL

The object model visualizes the elements in a software application in terms of objects. In this chapter, we will look into the basic concepts and terminologies of object–oriented systems.

### Objects and Classes

The concepts of objects and classes are intrinsically linked with each other and form the foundation of object–oriented paradigm.

Object

An object is a real-world element in an object–oriented environment that may have a physical or a conceptual existence. Each object has:

- Identity that distinguishes it from other objects in the system.
- State that determines the characteristic properties of an object as well as the values of the properties that the object holds.
- Behavior that represents externally visible activities performed by an object in terms of changes in its state.

Objects can be modeled according to the needs of the application. An object may have a physical existence, like a customer, a car, etc.; or an intangible conceptual existence, like a project, a process, etc.

A class represents a collection of objects having same characteristic properties that exhibit common behavior. It gives the blueprint or description of the objects that can be created from it. Creation of an object as a member of a class is called instantiation. Thus, object is an instance of a class.

The constituents of a class are:

- A set of attributes for the objects that are to be instantiated from the class. Generally, different objects of a class have some difference in the values of the attributes. Attributes are often referred as class data.
- A set of operations that portray the behavior of the objects of the class. Operations are also referred as functions or methods.

Let us consider a simple class, Circle, that represents the geometrical figure circle in a two–dimensional space. The attributes of this class can be identified as follows:

- x–coord, to denote x–coordinate of the center
- y–coord, to denote y–coordinate of the center
- a, to denote the radius of the circle
- findArea(), method to calculate area
- findCircumference(), method to calculate circumference
- scale(), method to increase or decrease the radius

During instantiation, values are assigned for at least some of the attributes. If we create an object my_circle, we can assign values like x-coord : 2, y-coord : 3, and a :4 to depict its state. Now, if the operation scale() is performed on my_circle with a scaling factor of 2, the value of the variable a will become 8. This operation brings a change in the state of my_circle, i.e., the object has exhibited certain behavior.

### Encapsulation and Data Hiding

Encapsulation

Encapsulation is the process of binding both attributes and methods together within a class. Through encapsulation, the internal details of a class can be hidden from outside. It permits the elements of the class to be accessed from outside only through the interface provided by the class.

### Data Hiding

Typically, a class is designed such that its data (attributes) can be accessed only by its class methods and insulated from direct outside access. This process of insulating an object's data is called data hiding or information hiding.

**Example**
In the class Circle, data hiding can be incorporated by making attributes invisible from outside the class and adding two more methods to the class for accessing class data, namely:

- setValues(), method to assign values to x-coord, y-coord, and a
- getValues(), method to retrieve values of x-coord, y-coord, and a

Here the private data of the object my_circle cannot be accessed directly by any method that is not encapsulated within the class Circle. It should instead be accessed through the methods setValues() and getValues().

Message Passing

Any application requires a number of objects interacting in a harmonious manner. Objects in a system may communicate with each other using message passing. Suppose a system has two objects: obj1 and obj2. The object obj1 sends a message to object obj2, if obj1 wants obj2 to execute one of its methods. The features of message passing are:

- Message passing between two objects is generally unidirectional.
- Message passing enables all interactions between objects.
- Message passing essentially involves invoking class methods.
- Objects in different processes can be involved in message passing.

Inheritance

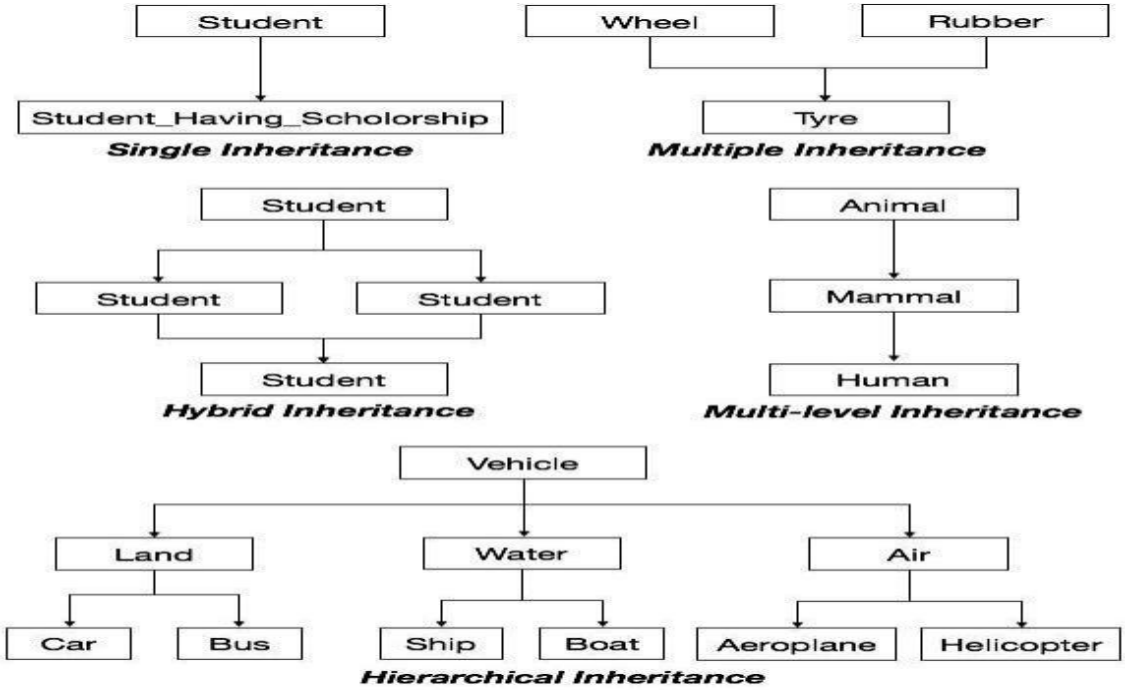
Inheritance is the mechanism that permits new classes to be created out of existing classes by extending and refining its capabilities. The existing classes are called the base classes/parent classes/super-classes, and the new classes are called the derived classes/child classes/subclasses. The subclass can inherit or derive the attributes and methods of the super-class(es) provided that the super-class allows so. Besides, the subclass may add its own attributes and methods and may modify any of the super-class methods. Inheritance defines an "is – a" relationship.

Example

From a class Mammal, a number of classes can be derived such as Human, Cat, Dog, Cow, etc. Humans, cats, dogs, and cows all have the distinct characteristics of mammals. In addition, each has its own particular characteristics. It can be said that a cow "is – a" mammal.
Types of Inheritance

- Single Inheritance: A subclass derives from a single super-class.
- Multiple Inheritance: A subclass derives from more than one super-classes.
- Multilevel Inheritance: A subclass derives from a super-class which in turn is derived from another class and so on.
- Hierarchical Inheritance: A class has a number of subclasses each of which may have subsequent subclasses, continuing for a number of levels, so as to form a tree structure.
- Hybrid Inheritance: A combination of multiple and multilevel inheritance so as to form a lattice structure.

The following figure depicts the examples of different types of inheritance.




**Polymorphism**

Polymorphism is originally a Greek word that means the ability to take multiple forms. In object-oriented paradigm, polymorphism implies using operations in different ways, depending upon the instance they are operating upon. Polymorphism allows objects with different internal structures to have a common external interface. Polymorphism is particularly effective while implementing inheritance.

**Example**

Let us consider two classes, Circle and Square, each with a method findArea(). Though the name and purpose of the methods in the classes are same, the internal implementation, i.e., the procedure of calculating area is different for each class. When an object of class Circle invokes its findArea() method, the operation finds the area of the circle without any conflict with the findArea() method of the Square class.

**Generalization and Specialization**

Generalization and specialization represent a hierarchy of relationships between classes, where subclasses inherit from super-classes.

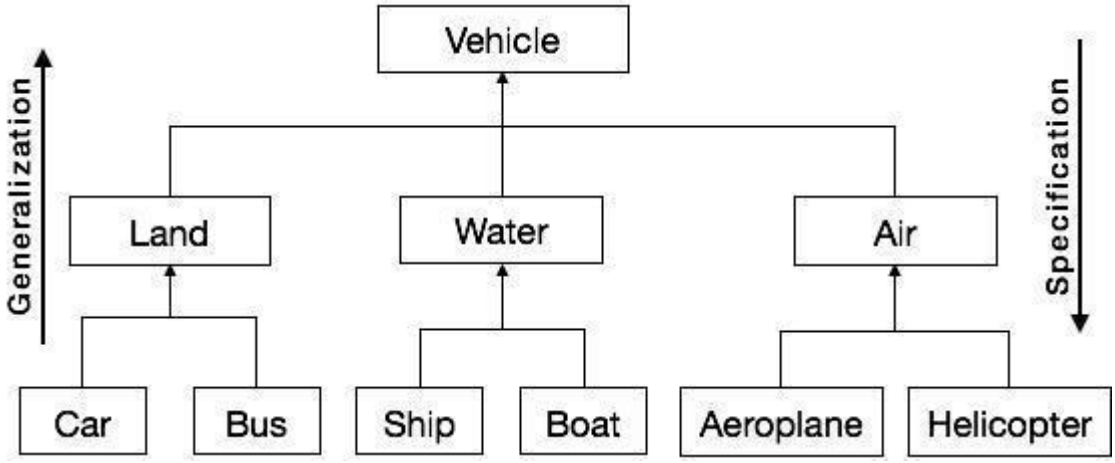
**Generalization**

In the generalization process, the common characteristics of classes are combined to form a class in a higher level of hierarchy, i.e., subclasses are combined to form a generalized super-class. It represents an “is – a – kind – of” relationship. For example, “car is a kind of land vehicle”, or “ship is a kind of water vehicle”.

**Specialization**

Specialization is the reverse process of generalization. Here, the distinguishing features of groups of objects are used to form specialized classes from existing classes. It can be said that the subclasses are the specialized versions of the super-class.

The following figure shows an example of generalization and specialization.




**Links and Association**

**Link**

A link represents a connection through which an object collaborates with other objects.

Rumbaugh has defined it as "a physical or conceptual connection between objects". Through a

link, one object may invoke the methods or navigate through another object. A link depicts the relationship between two or more objects.

**Association**

Association is a group of links having common structure and common behavior. Association depicts the relationship between objects of one or more classes. A link can be defined as an instance of an association.

**Degree of an Association**

Degree of an association denotes the number of classes involved in a connection. Degree may be unary, binary, or ternary.

- A unary relationship connects objects of the same class.
- A binary relationship connects objects of two classes.
- A ternary relationship connects objects of three or more classes.

**Cardinality Ratios of Associations**

Cardinality of a binary association denotes the number of instances participating in an association. There are three types of cardinality ratios, namely:

- One–to–One : A single object of class A is associated with a single object of class B.
- One–to–Many : A single object of class A is associated with many objects of class B.

- Many–to–Many : An object of class A may be associated with many objects of class B and conversely an object of class B may be associated with many objects of class A.

**Aggregation or Composition**

Aggregation or composition is a relationship among classes by which a class can be made up of any combination of objects of other classes. It allows objects to be placed directly within the body of other classes. Aggregation is referred as a "part–of" or "has–a" relationship, with the ability to navigate from the whole to its parts. An aggregate object is an object that is composed of one or more other objects.

**Example**

In the relationship, "a car has–a motor", car is the whole object or the aggregate, and the motor is a "part–of" the car. Aggregation may denote:

- Physical containment: Example, a computer is composed of monitor, CPU, mouse, keyboard, and so on.
- Conceptual containment: Example, shareholder has–a share.

Benefits of Object Model

Now that we have gone through the core concepts pertaining to object orientation, it would be worthwhile to note the advantages that this model has to offer. The benefits of using the object model are:

- It helps in faster development of software.
- It is easy to maintain. Suppose a module develops an error, then a programmer can fix that particular module, while the other parts of the software are still up and running.
- It supports relatively hassle-free upgrades.
- It enables reuse of objects, designs, and functions.
- It reduces development risks, particularly in integration of complex systems.

## OBJECT-ORIENTED SYSTEM

We know that the Object-Oriented Modeling (OOM) technique visualizes things in an application by using models organized around objects. Any software development approach goes through the following stages: Analysis, Design, and Implementation.

In object-oriented software engineering, the software developer identifies and organizes the application in terms of object-oriented concepts, prior to their final representation in any specific programming language or software tools.

**Phases in Object-Oriented Software Development**

The major phases of software development using object–oriented methodology are object-oriented analysis, object-oriented design, and object-oriented implementation.

**Object–Oriented Analysis**

In this stage, the problem is formulated, user requirements are identified, and then a model is built based upon real–world objects. The analysis produces models on how the desired system should function and how it must be developed. The models do not include any implementation details so that it can be understood and examined by any non–technical application expert.

**Object–Oriented Design**

Object-oriented design includes two main stages, namely, system design and object design.

System Design

In this stage, the complete architecture of the desired system is designed. The system is conceived as a set of interacting subsystems that in turn is composed of a hierarchy of interacting objects, grouped into classes. System design is done according to both the system analysis model and the proposed system architecture. Here, the emphasis is on the objects comprising the system rather than the processes in the system.

Object Design

In this phase, a design model is developed based on both the models developed in the system analysis phase and the architecture designed in the system design phase. All the classes required are identified. The designer decides whether:

- new classes are to be created from scratch,
- any existing classes can be used in their original form, or
- new classes should be inherited from the existing classes.

The associations between the identified classes are established and the hierarchies of classes are identified. Besides, the developer designs the internal details of the classes and their associations, i.e., the data structure for each attribute and the algorithms for the operations.

**Object–Oriented Implementation and Testing**

In this stage, the design model developed in the object design is translated into code in an appropriate programming language or software tool. The databases are created and the specific hardware requirements are ascertained. Once the code is in shape, it is tested using specialized techniques to identify and remove the errors in the code.

**Object-Oriented Principles**

**Principles of Object-Oriented Systems**

The conceptual framework of object–oriented systems is based upon the object model. There are two categories of elements in an object-oriented system:

**Major Elements** : By major, it is meant that if a model does not have any one of these elements, it ceases to be object oriented. The four major elements are:

- Abstraction
- Encapsulation
- Modularity
- Hierarchy

Minor Elements: By minor, it is meant that these elements are useful, but not indispensable part of the object model. The three minor elements are:

- Typing
- Concurrency
- Persistence

**Abstraction**

Abstraction means to focus on the essential features of an element or object in OOP, ignoring its extraneous or accidental properties. The essential features are relative to the context in which the object is being used.
Grady Booch has defined abstraction as follows:
*"An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer."*
**Example** : When a class Student is designed, the attributes enrolment_number, name, course, and address are included while characteristics like pulse_rate and size_of_shoe are eliminated, since they are irrelevant in the perspective of the educational institution.

**Encapsulation**
Encapsulation is the process of binding both attributes and methods together within a class. Through encapsulation, the internal details of a class can be hidden from outside. The class has methods that provide user interfaces by which the services provided by the class may be used.

**Modularity**
Modularity is the process of decomposing a problem (program) into a set of modules so as to reduce the overall complexity of the problem. Booch has defined modularity as:
***"Modularity is the property of a system that has been decomposed into a set of cohesive and loosely coupled modules."***
Modularity is intrinsically linked with encapsulation. Modularity can be visualized as a way of mapping encapsulated abstractions into real, physical modules having high cohesion within the modules and their inter–module interaction or coupling is low.

**Hierarchy**
In Grady Booch's words, "Hierarchy is the ranking or ordering of abstraction".

Through hierarchy, a system can be made up of interrelated subsystems, which can have their own subsystems and so on until the smallest level components are reached. It uses the principle of "divide and conquer". Hierarchy allows code reusability.

The two types of hierarchies in OOA are:
"IS–A" hierarchy : It defines the hierarchical relationship in inheritance, whereby from a super-class, a number of subclasses may be derived which may again have subclasses and so on. For example, if we derive a class Rose from a class Flower, we can say that a rose "is–a" flower.
"PART–OF" hierarchy : It defines the hierarchical relationship in aggregation by which a class may be composed of other classes. For example, a flower is composed of sepals, petals, stamens, and carpel. It can be said that a petal is a "part–of" flower.
**Typing**
According to the theories of abstract data type, a type is a characterization of a set of elements. In OOP, a class is visualized as a type having properties distinct from any other types. Typing is the enforcement of the notion that an object is an instance of a single class or type. It also enforces

that objects of different types may not be generally interchanged; and can be interchanged only in a very restricted manner if absolutely required to do so.
The two types of typing are:

- Strong Typing : Here, the operation on an object is checked at the time of compilation, as in the programming language Eiffel.
- Weak Typing : Here, messages may be sent to any class. The operation is checked only at the time of execution, as in the programming language Smalltalk.

Concurrency
Concurrency in operating systems allows performing multiple tasks or processes simultaneously. When a single process exists in a system, it is said that there is a single thread of control. However, most systems have multiple threads, some active, some waiting for CPU, some suspended, and some terminated. Systems with multiple CPUs inherently permit concurrent threads of control; but systems running on a single CPU use appropriate algorithms to give equitable CPU time to the threads so as to enable concurrency.

In an object-oriented environment, there are active and inactive objects. The active objects have independent threads of control that can execute concurrently with threads of other objects. The active objects synchronize with one another as well as with purely sequential objects.

Persistence

An object occupies a memory space and exists for a particular period of time. In traditional programming, the lifespan of an object was typically the lifespan of the execution of the program that created it. In files or databases, the object lifespan is longer than the duration of the process creating the object. This property by which an object continues to exist even after its creator ceases to exist is known as persistence.

**Object-Oriented Analysis**
In the system analysis or object-oriented analysis phase of software development, the system requirements are determined, the classes are identified and the relationships among classes are identified. The three analysis techniques that are used in conjunction with each other for object-oriented analysis are object modeling, dynamic modeling, and functional modeling.

**Object Modeling**
Object modeling develops the static structure of the software system in terms of objects. It identifies the objects, the classes into which the objects can be grouped into and the relationships between the objects. It also identifies the main attributes and operations that characterize each class. The process of object modeling can be visualized in the following steps:

- Identify objects and group into classes
- Identify the relationships among classes
- Create user object model diagram
- Define user object attributes
- Define the operations that should be performed on the classes
- Review glossary

After the static behavior of the system is analyzed, its behavior with respect to time and external changes needs to be examined. This is the purpose of dynamic modelling. Dynamic modelling can be defined as "a way of describing how an individual object responds to events, either internal events triggered by other objects, or external events triggered by the outside world".
The process of dynamic modelling can be visualized in the following steps:
Identify states of each object

- Identify events and analyze the applicability of actions
- Construct dynamic model diagram, comprising of state transition diagrams
- Express each state in terms of object attributes
- Validate the state–transition diagrams drawn

**Functional Modelling**

Functional Modelling is the final component of object-oriented analysis. The functional model shows the processes that are performed within an object and how the data changes as it moves between methods. It specifies the meaning of the operations of object modelling and the actions of dynamic modelling. The functional model corresponds to the data flow diagram of traditional structured analysis.

The process of functional modelling can be visualized in the following steps:

- Identify all the inputs and outputs
- Construct data flow diagrams showing functional dependencies
- State the purpose of each function
- Identify constraints
- Specify optimization criteria

**Structured Analysis vs. Object-Oriented Analysis**

The Structured Analysis/Structured Design (SASD) approach is the traditional approach of software development based upon the ***waterfall model***. The phases of development of a system using SASD are:

- Feasibility Study
- Requirement Analysis and Specification
- System Design
- Implementation
- Post-implementation Review

Now, we will look at the relative advantages and disadvantages of structured analysis approach and object-oriented analysis approach.

**Advantages/Disadvantages of Object-Oriented Analysis**

| Advantages | Disadvantages |
|---|---|
| Focuses on data rather than the procedures as in Structured Analysis. | Functionality is restricted within objects. This may pose a problem for systems which are intrinsically procedural or computational in nature. |

| | |
|---|---|
| The principles of encapsulation and data hiding help the developer to develop systems that cannot be tampered by other parts of the system. | It cannot identify which objects would generate an optimal system design. |
| The principles of encapsulation and data hiding help the developer to develop systems that cannot be tampered by other parts of the system. | The object-oriented models do not easily show the communications between the objects in the system. |
| It allows effective management of software complexity by the virtue of modularity. | All the interfaces between the objects cannot be represented in a single diagram. |
| It can be upgraded from small to large systems at a greater ease than in systems following structured analysis. | |

| Advantages | Disadvantages |
|---|---|
| As it follows a top-down approach in contrast to bottom-up approach of object-oriented analysis, it can be more easily comprehended than OOA. | In traditional structured analysis models, one phase should be completed before the next phase. This poses a problem in design, particularly if errors crop up or requirements change. |
| It is based upon functionality. The overall purpose is identified and then functional decomposition is done for developing the software. The emphasis not only gives a better understanding of the system but also generates more complete systems. | The initial cost of constructing the system is high, since the whole system needs to be designed at once leaving very little option to add functionality later |
| The specifications in it are written in simple English language, and hence can be more easily analyzed by non-technical personnel. | It does not support reusability of code. So, the time and cost of development is inherently high. |

**States and State Transitions**

**State**

The state is an abstraction given by the values of the attributes that the object has at a particular time period. It is a situation occurring for a finite time period in the lifetime of an object, in which it fulfils certain conditions, performs certain activities, or waits for certain events to occur. In state transition diagrams, a state is represented by rounded rectangles.

**Parts of a State**

- Name : A string differentiates one state from another. A state may not have any name.
- Entry/Exit Actions : It denotes the activities performed on entering and on exiting the state.
- Internal Transitions : The changes within a state that do not cause a change in the state.
- Sub–states : States within states.

**Initial and Final States**

The default starting state of an object is called its initial state. The final state indicates the completion of execution of the state machine. The initial and the final states are pseudo-states, and may not have the parts of a regular state except name. In state transition diagrams, the initial state is represented by a filled black circle. The final state is represented by a filled black circle encircled within another unfilled black circle.

**Transition**

A transition denotes a change in the state of an object. If an object is in a certain state when an event occurs, the object may perform certain activities subject to specified conditions and change the state. In this case, a state−transition is said to have occurred. The transition gives the relationship between the first state and the new state. A transition is graphically represented by a solid directed arc from the source state to the destination state.

**The five parts of a transition are**:

Source State: The state affected by the transition.

Event Trigger: The occurrence due to which an object in the source state undergoes a transition if the guard condition is satisfied.

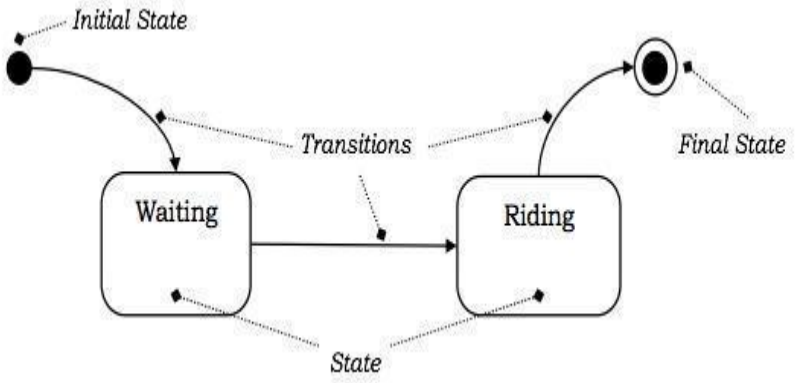
Guard Condition: A Boolean expression which if True, causes a transition on receiving the event trigger.

Action: An un-interruptible and atomic computation that occurs on the source object due to some event.

Target State : The destination state after completion of transition.

**Example**

Suppose a person is taking a taxi from place X to place Y. The states of the person may be: Waiting (waiting for taxi), Riding (he has got a taxi and is travelling in it), and Reached (he has reached the destination). The following figure depicts the state transition.




**Events**

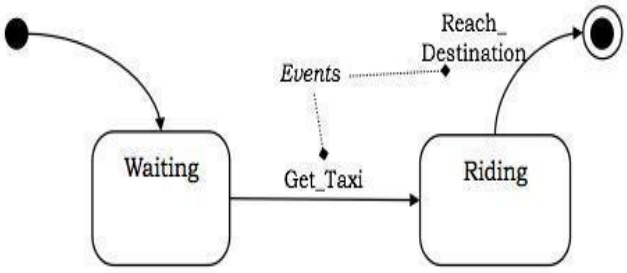
Events are some occurrences that can trigger state transition of an object or a group of objects. Events have a location in time and space but do not have a time period associated with it. Events are generally associated with some actions.

Examples of events are mouse click, key press, an interrupt, stack overflow, etc.

Events that trigger transitions are written alongside the arc of transition in state diagrams.

**Example**

Considering the example shown in the above figure, the transition from Waiting state to Riding state takes place when the person gets a taxi. Likewise, the final state is reached, when he reaches the destination. These two occurrences can be termed as events Get_Taxi and Reach_Destination. The following figure shows the events in a state machine.




**External and Internal Events**

External events are those events that pass from a user of the system to the objects within the system. For example, mouse click or key−press by the user are external events. Internal events are those that pass from one object to another object within a system. For example, stack overflow, a divide error, etc.

**Deferred Events**

Deferred events are those which are not immediately handled by the object in the current state but are lined up in a queue so that they can be handled by the object in some other state at a later time.

**Event Classes**

Event class indicates a group of events with common structure and behavior. As with classes of objects, event classes may also be organized in a hierarchical structure. Event classes may have attributes associated with them, time being an implicit attribute. For example, we can consider the events of departure of a flight of an airline, which we can group into the following class:

Flight_Departs (Flight_No, From_City, To_City, Route)

**Activity**

Activity is an operation upon the states of an object that requires some time period. They are the ongoing executions within a system that can be interrupted. Activities are shown in activity diagrams that portray the flow from one activity to another.

Action

An action is an atomic operation that executes as a result of certain events. By atomic, it is meant that actions are un-interruptible, i.e., if an action starts executing, it runs into completion without being interrupted by any event. An action may operate upon an object on which an event has been triggered or on other objects that are visible to this object. A set of actions comprise an activity.

**Entry and Exit Actions**

Entry action is the action that is executed on entering a state, irrespective of the transition that led into it.

Likewise, the action that is executed while leaving a state, irrespective of the transition that led out of it, is called an exit action.

Scenario

Scenario is a description of a specified sequence of actions. It depicts the behavior of objects undergoing a specific action series. The primary scenarios depict the essential sequences and the secondary scenarios depict the alternative sequences.

**Diagrams for Dynamic Modeling**

There are **two** primary diagrams that are used for dynamic modeling:

**Interaction Diagrams**

Interaction diagrams describe the dynamic behavior among different objects. It comprises of a set of objects, their relationships, and the message that the objects send

and receive. Thus, an interaction models the behavior of a group of interrelated objects.
The two types of interaction diagrams are:

- Sequence Diagram : It represents the temporal ordering of messages in a tabular manner.
- Collaboration Diagram : It represents the structural organization of objects that send and receive messages through vertices and arcs.

**State Transition Diagram**

State transition diagrams or state machines describe the dynamic behavior of a single object. It illustrates the sequences of states that an object goes through in its lifetime, the transitions of the states, the events and conditions causing the transition and the responses due to the events.

Concurrency of Events

In a system, two types of concurrency may exist. They are discussed below.

System Concurrency

Here, concurrency is modelled in the system level. The overall system is modelled as the aggregation of state machines, where each state machine executes concurrently with others.

Concurrency within an Object

Here, an object can issue concurrent events. An object may have states that are composed of sub-states, and concurrent events may occur in each of the sub-states.

Concepts related to concurrency within an object are as follows:

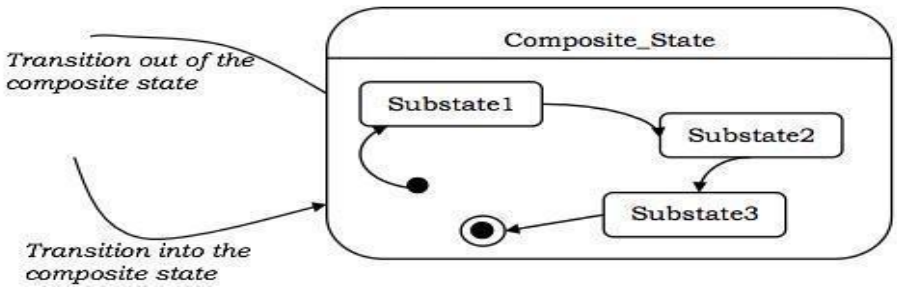
Simple and Composite States

A simple state has no sub-structure. A state that has simpler states nested inside it is called a composite state. A sub-state is a state that is nested inside another state. It is generally used to reduce the complexity of a state machine. Sub-states can be nested to any number of levels. Composite states may have either sequential sub-states or concurrent sub-states.

Sequential Sub-states

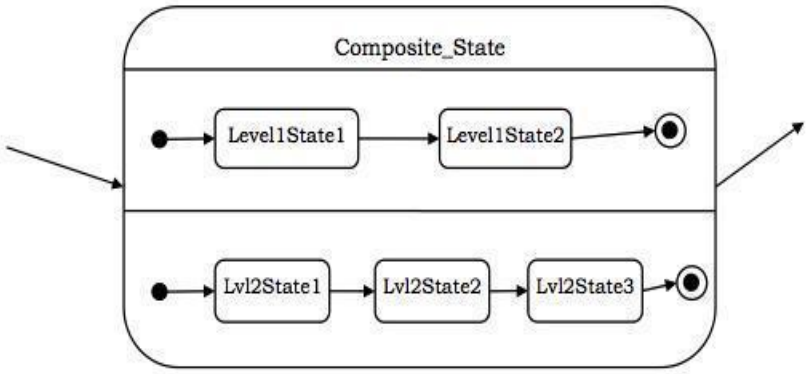
In sequential sub-states, the control of execution passes from one sub-state to another sub-state one after another in a sequential manner. There is at most one initial state and one final state in these state machines.

The following figure illustrates the concept of sequential sub-states.



**Concurrent Sub-states**

In concurrent sub-states, the sub-states execute in parallel, or in other words, each state has concurrently executing state machines within it. Each of the state machines has its own initial and final states. If one concurrent sub-state reaches its final state before the other, control waits at its final state. When all the nested state machines reach their final states, the sub-states join back to a single flow.
The following figure shows the concept of concurrent sub-states.

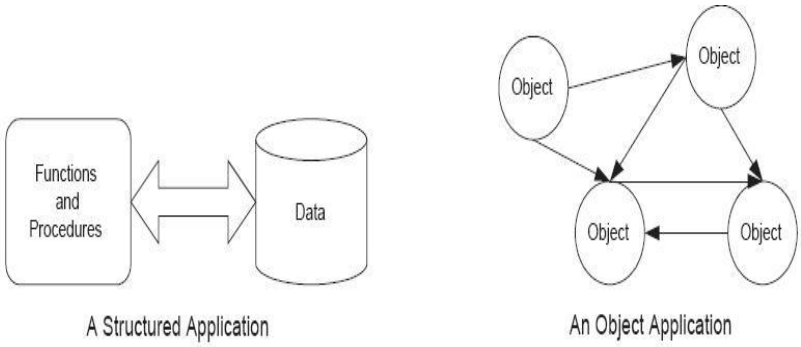



**Structured vs. Object Orientation paradigm**

Structured paradigm

- The structured *paradigm is* a development strategy based on the concept that a system should be separated into two parts:
    - Data and functionality (modeled using a process model). Using the structured approach, you develop applications in which data is separated from behavior in both the design model and in the system implementation (that is, the program).
- Example: Consider the design of an information system for a university. Taking the structured approach, you would define the layout of a data initially as a separate system and the design of a program to access that data as another. The programs have the ability to change the data states.

- The main concept behind the object-oriented paradigm is that instead of defining systems as two separate parts (data and functionality), system defined as a collection of interacting objects.
    - *Describes and build system that consists object*
- An object-oriented system comprises a number of software objects that interact to achieve the system objective.




The Potential Benefits of the Object Oriented paradigm

Increased reusability:

The OO paradigm provides opportunities for reuse through the concepts of inheritance, polymorphism, encapsulation, modularity, coupling and cohesion. It provides more opportunities for reuse than the structured paradigm
Increased extensibility
Because classes have both data and functionality, when you add new features to the system you need to make changes in one place, the class;
Improved Quality
Quality systems are on time, on budget and meet or exceed the expectations of their users.
Improved quality comes from increased participation of users in systems development.
OO systems development techniques provide greater opportunity for users to participate in the development process.

Reusability, extensibility, and improved quality are all technical benefits. Object orientation enables you to build systems better, faster and cheaper (BFC). The benefits OO are realized through out the entire development life cycle, not just programming

Increased Chance of Project success

- A project is successful if it is on time, on budget and meets the needs of the its users.
- Users are expert at business and they are the only ones who can tell you what they need.
- You need to know the right question to ask, know the business very well.

- You need models that communicate the required information and that users understand.
- You need to work closely with users
- Time invested in defining requirements and modeling pays off in the long run.
- You can use a wide variety of artifacts including code, model and components.

- Software organizations currently spend significant resources (80%) maintaining and operating software, and because of the long waiting list of work to be done, it takes significant time to get new projects started. These two problems are respectively called
  - the maintenance Burdon and
  - The application backlog
- These are problems that object orientation can help you to overcome

The Potential Drawbacks of OO

Nothing is perfect including OO. While many exiting benefits exist to OO, they come at a price:

1. OO requires greater concentration on requirements analysis and design
   - You cannot build a system that meets users needs unless you know what those needs are( you need to do requirements)
   - You cannot built a system unless you know how it all fit together (you need to do analysis and design)
   - But this fact is often ignored by many developers
2. Developers must closely work with users
   - Users are the experts but they have their own jobs to do (busy)
3. OO requires a complete change in the mindset on the part of individuals
   - they should understand the benefits of OO
4. OO requires the development culture of the IS dept to change
   - The change in the mind set of individual developers actually reflect an over all change in the development culture
   - Do more analysis and design but (less programming) and working with users
5. OO is just more than programming
6. Many OO benefits are long term
   - OO truly pays off when you extend and enhance your system
7. OO demands up front investments in training education and tools
   - Organizations must train and educate their development staff.
   - Buy books, development tools and magazines
8. OO techniques do not guarantee you will build the right system
   - While OO increases the probability of project success, it still depends on the ability of individuals involved.
   - Developers, users, mangers must be working together to have a good working atmosphere.
9. OO necessitates increased testing
   - OO is typically iterative in nature, and probably developing complex system using the objects, the end result is you need to spend more time in testing.
10. OO is only part of the solution
    - You still need CASE tools
    - Need to perform quality assurance (QA)
    - You still need usable interface so the users can work with the systems effectively

**Object Standards**

- OO orientation today becomes the significant part of the software development.
- Objects are the primary enabling technology for components.
- It also stays in the future because of the standard set by the OMG.
- OO orientation today becomes the significant part of the software development.
- Objects are the primary enabling technology for components.
- It also stays in the future because of the standard set by the OMG.
    - **CORBA**( Common object request broker architecture) – the standard architecture for supporting distributed objects.
    - **UML** ( Unified modeling language)-the standard modeling language for the object oriented software.
    - **ANSI** ( Americans National Standards Institute)-Defined standards for C++. Http://www.ansi.org
    - **Sun Microsystems** ,Http://www.sum.com actively maintains, enhances and supports a de facto standard definition for java and related standards such as Enterprise Java Beans (EJB).
    - The Object Database Management group (**ODMG**)-Http://www.odmg.org actively maintains, enhances and supports a standard definition for object oriented databases and object query language (OQL).
    - **ANSI** ( Americans National Standards Institute)-Defined standards for C++. Http://www.ansi.org
    - **Sun Microsystems** ,Http://www.sum.com actively maintains, enhances and supports a de facto standard definition for java and related standards such as Enterprise Java Beans (EJB).
    - **The Object Database Management group (ODMG**)-Http://www.odmg.org actively maintains, enhances and supports a standard definition for object oriented databases and object query language (OQL).

**Object-Oriented Programming Principles**

UML is the international standard notation for object-oriented analysis and design. It is defined by the Object Management Group (www.omg.org) and is currently at release 1.4 with 2.0 expected next year. UML is a sound basis for object-oriented methods including those that apply to component based development. This section focuses both on the widely used UML notation and upon the principles of modeling. UML is a standardized notation for object-oriented analysis and design. However, a method is more than a notation. To be an analysis or design method it must include guidelines for using the notation and methodological principles. To be a complete software engineering method it must also include procedures for dealing with matters outside the scope of mere software development: business and requirements modeling, development process, project management, metrics, traceability techniques and reuse management. In this tutorial we focus on the notational and analysis and design aspects.

**The history of object-oriented analysis and design methods**

The development of computer science as a whole proceeded from an initial concern with programming alone, through increasing interest in design, to concern with analysis methods only latterly. Reflecting this perhaps, interest in object-orientation also began, historically, with language developments. It was only in the 1980s that object-oriented design methods emerged. Object-oriented analysis methods emerged during the 1990s.
Apart from a few fairly obscure AI applications, up until the 1980s object orientation was largely associated with the development of graphical user interfaces (GUIs) and few other applications became widely known. Up to this period not a word had been mentioned about analysis or design for object-oriented systems.
With the 1990s came both increased pressures on business and the availability of cheaper and much more powerful computers. This led to a ripening of the field and to a range of applications beyond GUIs and AI. Distributed, open computing became both possible and important and object technology was the basis of much development, especially with the appearance of n-tier client-server systems and the web, although relational databases played and continue to play an important role.

The new applications and better hardware meant that mainstream organizations adopted object-oriented programming and now wanted proper attention paid to object-oriented design and (next) analysis. Concern shifted from design to analysis, from the start of the 1990s. An object-oriented approach to requirements engineering had to wait even longer.

The benefits of object-oriented analysis and design specifically include:

- required changes are localized and unexpected interactions with other program modules are unlikely;
- inheritance and polymorphism make OO systems more extensible, contributing thus to more rapid development;
- object-based design is suitable for distributed, parallel or sequential implementation;
- objects correspond more closely to the entities in the conceptual worlds of the designer and user, leading to greater seamlessness and traceability;

- shared data areas are encapsulated, reducing the possibility of unexpected modifications or other update anomalies.

**Object-oriented analysis and design using UML**

The Unified Modeling Language (UML) is probably the most widely known and used notation for object-oriented analysis and design. It is the result of the merger of several early contributions to object-oriented methods. It will be illustrated how to go about object-oriented analysis and design in UML.

The UML includes specifications for nine different diagrams used to document various perspectives of a software solution from project inception to installation and maintenance. In addition, Packages provides a means to organize work. The Component and Deployment diagrams describe an implementation. The remaining seven diagrams are used to model requirements and design. This section presents a way to organize these last seven diagrams to make them easier to remember and apply, and an overview of the principles that guide their development.

**Views**

One way to organize the UML diagrams is by using views. A *view* is a collection of diagrams that describe a similar aspect of the project. This section uses a set of three distinct yet complementary views that are called the Static View, Dynamic View, and Functional View.

Together, the three views provide a complete and overlapping picture of the subject any modeling process.

**Functional View**

The Functional View includes both the Use Case diagram and the Activity diagram. They are kept together because they are used together so often and they both model how the system is supposed to work. Figure 1.0 below shows that the *Use Case diagram* defines the functions that the system must provide. The functions are expressed first as goals. But then the goals are fleshed out in a narrative to describe what each Use Case is expected to do to achieve each goal.

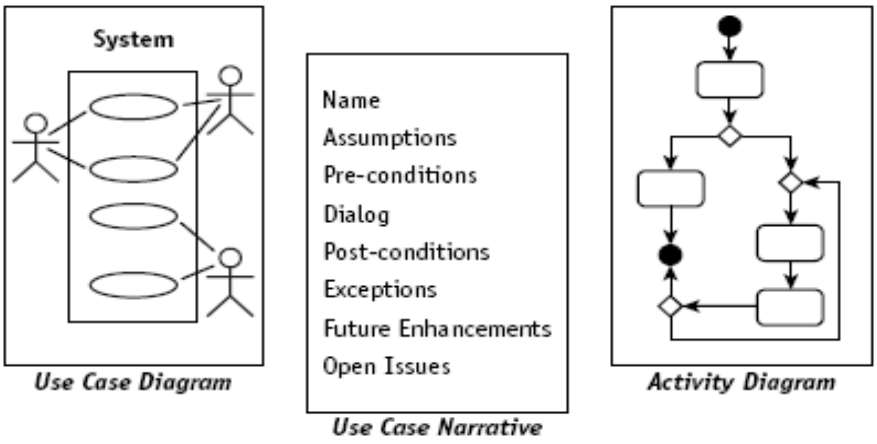



**Figure 1.0**

This Use Case description can be written, but the logic is drawn with an *Activity diagram*, A diagram that models logic very much like a flowchart. The Activity diagram is also useful for modeling a workflow or a business process. It can be very useful when working with clients to determine how things are or should be done. You can also use the Activity diagram to assess the complexity of the application, and to verify the internal consistency of your Use Case definitions. Later in the design, you need to specify the implementation details for methods. When the details become complicated, being able to draw them out using the Activity diagram makes the logic much easier to understand.

- The **Use Case diagram** describes the features that the users expect the system to provide.
- The **Activity diagram** describes processes including sequential tasks, conditional logic, and concurrency. This diagram is like a flowchart, but it has been enhanced for use with object modeling.

**Static View**

The Static View includes those diagrams that provide a snapshot of the elements of the system but don't tell you how the elements will behave. It is very much like a blueprint. Blueprints are comprehensive, but they only show what remains stationary, hence the term
*Static View.* Figure 1.1 below illustrates the two diagrams that make up the Static View, the Class diagram and the Object diagram. The *Class diagram* is the primary tool of the Static View. It provides a fixed look at every resource (class) and its features. It is the one diagram nearly always used for code generation and reverse engineering.

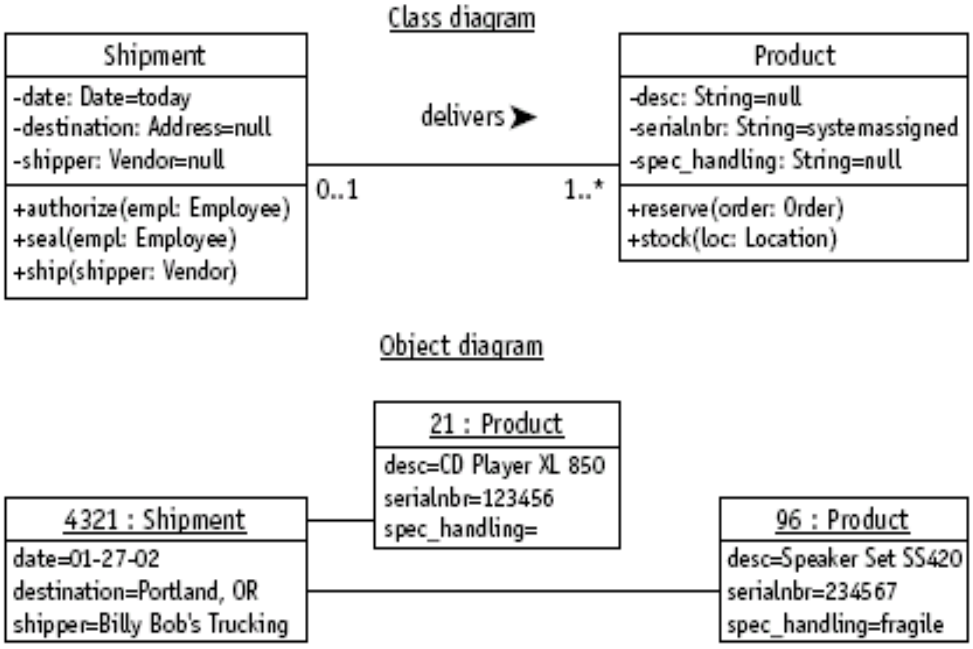



**Figure 1.1**

The Object diagram is also used to test the Class diagram. Class diagrams are very easy to draw, and even easier to draw badly. Logical processes create objects and their relationships. Tracing the affects of a process on an Object diagram can reveal problems in the logic of the process. Drawing Object diagrams for test data can be just as effective as drawing them for examples. Hard facts will either confirm or deny the accuracy of your Class diagram.

The **Class diagram** is the primary static diagram. It is the foundation for modeling the rules about types of objects (classes), the source for code generation, and the target for reverse engineering.

The **Object diagram** illustrates facts in the form of objects to model examples and test data. The Object diagram can be used to test or simply to understand a Class diagram.

**Dynamic View**

The figure below shows the three types of diagrams that make up the Dynamic View. The Dynamic View includes the diagrams that reveal how objects interact with one another in response to the environment. It includes the *Sequence and Collaboration diagrams,* which collectively are referred to as interaction diagrams. They are specifically designed to describe how objects talk to each other. It also includes the *Statechart diagram,* which shows how and why an object changes over time in response to the environment.

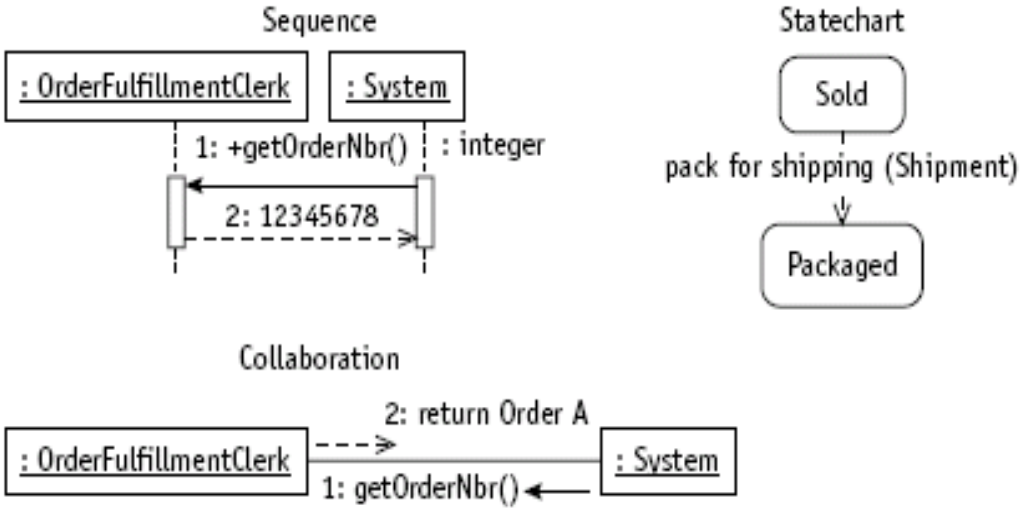



Figure 1.2

- For modeling object interactions, the Dynamic View includes the **Sequence and Collaboration diagrams**.
- The **Statechart diagram** provides a look at how an object reacts to external stimuli and manages internal changes.

**Object-Oriented Principles**

This section describes some of the principles that drive most of the UML modeling. All the UML diagrams describe some form of object-oriented information. But what does the term *object-oriented* mean? It views the things in the world as objects.
So what is an *object?* The simplest definition of an object is pretty much anything you can talk about. An object can be a physical entity, like the things you see, but an object may also be intangible, for example, a concept like an illness, attendance, or a job. Even though a job is not something you can touch, it is something you can describe, discuss, assign, and complete.

**Abstraction**

A software object is an *abstraction,* a representation of something in the real world. An abstraction is a way to describe something where you only include the information about it that is important to you.
The working definition for creating an abstraction is: *representing something in the real world in a useful manner to solve a specific problem.* Usefulness is measured by how well it helps you solve the problem you are trying to solve. The representation is an *object.* The rules that define the representation make up a *class.*
For comparison, think of the word *apple* in a dictionary and an *apple* in your hand. In software, the definition in the dictionary is a class, and the apple in your hand is an object. To make objects from classes, you must use the class definition like a template or a mold. Although each object may vary somewhat, all objects of the same class must conform to the class definition.

This is why an object can be an *instance* of a class. An object is created, manufactured, or instantiated (made real) from the class definition.

**Information**

An object should have its own current condition. This condition is formally called the *state* of the object. Simply put, the state of an object is a description of the properties of the object during a particular period of time. Consequently, when any of the properties of the object change, the state of the object is said to change. An object can describe itself and has known current condition (or state).

**Behaviour**

An object must know *what can be done to it.* There are a lot of people who can write with the pencil, including students, teachers, programmers, designers, and analysts. To do so, every class of objects in this list would have to include a description of the "write" behaviour. There are a lot of problems with this approach, including redundant maintenance, possible conflict between the different class definitions for how to "write" resulting in possible misuse of the pencil, and the fact that all these objects would need to be notified any changes to the pencil that would affect their ability to use their "write" behavior.

- An object knows what it can do
- An object knows what can be done to it

A better solution is to write one definition for the behavior in the pencil class. That way, anyone who wants to use the pencil to write goes to the pencil to find out how. Everyone gets the same implementation of the behaviour, and there is only one place to make changes.

**Encapsulation**

Encapsulation provides the means to organize this information so that you can use it and maintain it efficiently. Encapsulation separates the object into two categories:

- What you need to know in order to use the object
- What you need to know in order to make the object work properly

**To use the object**

An object must have an interface to be used.

**To make the object work properly**

An *interface* without an *implementation* doesn't do much. You could communicate with the object, but the object didn't have any way to respond. In order to make the object work, you need to provide the mechanisms that respond to the interface. An object knows about itself.

Consequently, encapsulation tells you that the information has to be inside the object with the behavior so that you can control access to it and protect its integrity. This is why encapsulation is often called *information hiding.* You hide the information inside the object, where the object has complete control. In order to make the object work properly, you need to place inside the object:

- The *implementations* for each interface
- The data that describes the *structure* of the object
- The data that describes the current *state* of the object

# THE COPPERBELT UNIVERSITY

COMPUTER SCIENCE DEPARTMENT

**CS 432 – OBJECT ORIENTED SYSTEMS ANALYSIS AND DESIGN**

COMPILED BY DR NTALASHA

## Object-Oriented Overview

Object-oriented techniques work well in situations where complicated systems are undergoing continuous maintenance, adaptation, and design. There are two ways to model object-oriented systems; Coad and Yourdon methodology and The Unified Modeling Language.

### Object-Oriented Programming

There are six ideas characterize object-oriented programming:

- An object, which represents a real-world thing or event
- A class, or group of related objects
- Messages, sent between objects
- Encapsulation, only an object makes changes through its own behavior
- Inheritance, a new class created from another class
- Polymorphism, meaning that a derived class behavior may be different from the base class

### Five-Layer Model

The Object oriented analysis and design is based on a five-layer model;

- Class/object layer notes the classes and objects
- Structure layer captures various structures of classes and objects, such as one-to-many relationships and inheritance
- Attribute layer details the attributes of classes
- Service layer notes messages and object behaviors
- Subject layer divides the design into implementation units or team assignments

### Five General Types of Objects

There are five general types of objects:

- Tangible things
- Roles
- Incidents
- Interactions
- Specifications details

### Criteria to Determine Need for a New Class of Objects

The following are the criteria to determine whether a new class of objects is justified

- There is a need to remember the object
- There is a need for certain behaviors of the object
- An object has multiple attributes
- A class has more than one object instantiation unless it is a base class
- Attributes have a meaningful value for each object in a class
- Services behave the same for every object in a class
- Objects implement requirements that are derived from the problem setting
- Objects do not duplicate attributes and services that could be derived from other objects in the system

**Basic Types of Structures**

There are two basic types of structures that might be imposed on classes and objects: Generalization-Specialization structure (Gen-Spec), which connect class-to-class and Whole-Part structure which are collections of different objects that compose another whole object.

**Instance Connections**

Instance connections are references between objects such as associations or relationships indicated by a single line between objects using the same cardinality notation as Whole-Part structures.

**Methods**

Services (or methods or procedures) must be analyzed. Activities are

- Object state analysis, showing changes of state
- Service specification: creating, storing, retrieving, connecting, accessing, and deleting objects
- Message specification, consisting of control and data flow

**Major Components of Object-Oriented Design Activities**

The Object-oriented design activities are grouped into four major components:

- The problem domain component
- The human interface
- Major Components of Object-Oriented Design Activities component
- The data management component
- The task management component

**Problem Domain Component**

The problem domain component consists of Reuse design, Implementation structures and Language accommodation.

CRC Cards

Class, responsibilities, and collaborators (CRC) cards are used to represent the responsibilities of classes and the interaction between the classes.

Creating CRC Cards

Analysts create CRC cards by

- Finding all the nouns and verbs in a problem statement
- Create scenarios that are actually walkthroughs of system functions
- Identify and refine responsibilities into smaller and smaller tasks, if possible
- The group determines how tasks are fulfilled by objects or interacting with other things
- Responsibilities evolve into methods or operations

# THE COPPERBELT UNIVERSITY

COMPUTER SCIENCE DEPARTMENT

**CS 432 – OBJECT ORIENTED SYSTEMS ANALYSIS AND DESIGN**

COMPILED BY DR NTALASHA

## Understanding the Use Case Model

The Use Case *model* is a collection of diagrams and text that together document how users expect to interact with the system. Figure 1.0 illustrates the Use Case diagram, the Use Case narrative, and the Use Case scenarios (using a flowchart or Activity diagram). The Use Case model focuses on the critical success factors of the system, in terms of the functionality or features that the users need to interact with. By focusing on the system's features, you create a set of conceptual slots into which all the widely varied requirements can be placed. Features can be tested, modeled, designed, and implemented. Users who require a particular feature become the audience for the modeling activities for that feature. By focusing on features, you also define the scope of the project (that is, which features will and will not be supported by the final solution and, typically, the list of features that will be implemented in each incremental release.

## The Purpose of the Use Case Model

The key difference between Use Cases and functional design is the focus. Functional design documents a process, but a Use Case focuses on the goal of a process. This change in mindset is essential in keeping us from jumping to solutions without first understanding why.
Focusing on the process often leads to reproducing existing systems, rather than redesigning them, precisely because it focuses on "how" rather than "why." Goal-focused modeling keeps you focused on a target rather than the means of getting to the target. This keeps you open to a variety of solutions, allowing and possibly encouraging you to take advantage of technological advances.

## Use Case diagram

The Use Case diagram consists of five very simple graphics that represent the system, actors, Use Cases, associations, and dependencies of the project. The goal of the diagram is to provide a high-level explanation of the relationship between the system and the outside world. It is a very *flat* diagram (that is, it provides only a surface level, or black-box, view of the system).
The view represented by a Use Case diagram for an ATM application, for example, would correspond to the main screen of an ATM and the menu options available at that level. The ATM *system* offers the user a set of options such as withdraw, deposit, inquire on balance, and transfer funds. Each option can be represented by a separate Use Case. The customer (outside the system) is associated with each of the Use Cases (within the system) that he plans to use.

## Use Case narrative

On the Use Case diagram, a Use Case is simply an ellipse with a simple label like "Receive Product." Although this label may provide a meaningful interface, it doesn't explain what you can expect from this system feature. For that, you need a textual description. The Use Case narrative provides a fairly standard (yet user-defined) set of information that is required to guide the analysis, design, and coding of the feature.

**Use Case scenarios**

A Use Case scenario is one logical path through a Use Case, one possible sequence of steps in the execution of the Use Case. A Use Case may include any number of scenarios. The set of scenarios for one Use Case identifies everything that can happen when that Use Case is used. Consequently, the set of scenarios becomes the basis for your test plan for the Use Case. As the application design deepens, the test plans are expanded to keep the tests focused on the original expectations for the Use Case expressed in the scenarios. These three elements - the Use Case diagram, narrative, and scenarios - comprise the Use Case Model.

**Defining the Elements of the Use Case Diagram**

Of the three elements that comprise the Use Case Model, the only one actually defined by UML is the Use Case Diagram. Figure 1.3 shows a Use Case Diagram.

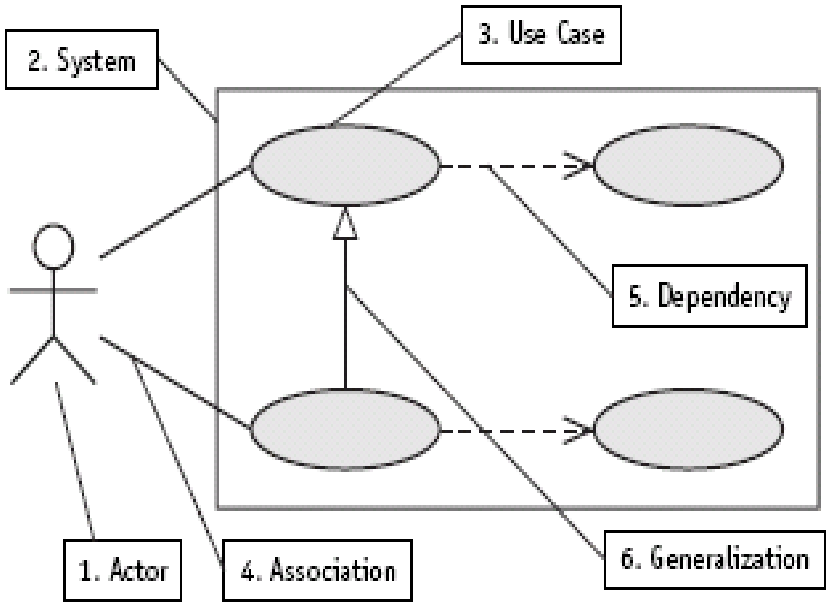



Figure 1.3

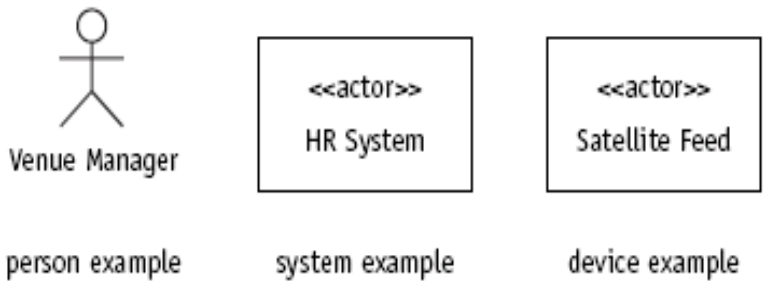
Six modeling elements make up the Use Case diagram: systems, actors, Use Cases, associations, dependencies, and generalizations.

- **System:** Sets the boundary of the system in relation to the actors who use it (outside the system) and the features it must provide (inside the system).

- **Actor:** A role played by a person, system, or device that has a stake in the successful operation of the system.
- **Use Case:** Identifies a key feature of the system. Without these features, the system will not fulfill the user/actor requirements. Each Use Case expresses a goal that the system must achieve.
- **Association:** Identifies an interaction between actors and Use Cases. Each association becomes a dialog that must be explained in a Use Case narrative. Each narrative in turn provides a set of scenarios that function as test cases when evaluating the analysis, design, and implementation of the Use Case.
- **Dependency:** Identifies a communication relationship between two Use Cases.
- **Generalization:** Defines a relationship between two actors or two Use Cases where one Use Case inherits and adds to or overrides the properties of the other.

**Use Case actors**

Systems always have users. Users in the classic sense are people who use the system. But users can also be other systems or devices that trade information. In Use Case diagrams, people, systems, and devices are all referred to as *actors.* The icons to model them may vary, but the concept remains the same. An actor is a *role* that an external entity plays in relation to the system. To reiterate, an actor is a *role,* not necessarily a particular person or a specific system. Figure 1.4 shows some actor icons.




**Use Cases**

Use Cases define the required features of the system. Without these features, the system cannot be used successfully. Each Use Case is named using a verb phrase that expresses a goal the system must accomplish, for example, deposit money, withdraw money, and adjust account. Although each Use Case implies a supporting process, the focus is on the goal, not the process. Below are some of *Use Case notation for the Use Case diagram.*

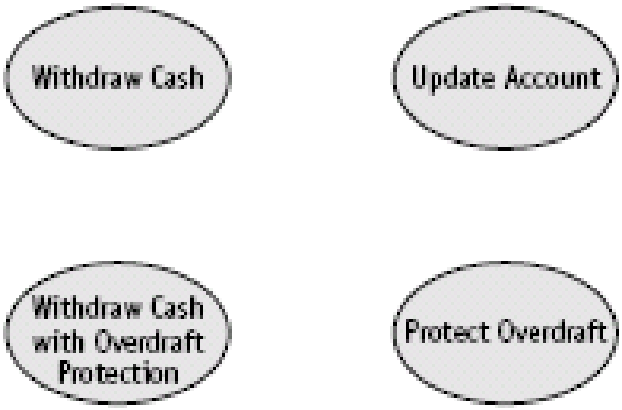



Figure showing Use Cases

By defining Use Cases in this manner, the system is defined as a set of requirements rather than a solution. You do not describe how the system must work. You describe what the system must be able to do. The Use Cases describe only those features visible and meaningful to the actors who use the system. Keeping this in mind will help you avoid *functional decomposition,* the breaking down of procedures and tasks into smaller and smaller processes until you have described all the internal workings of the system. One of the pitfalls of systems development is going over budget, which happens when you don't limit the scope of each task or you make a model too inclusive. The UML provides seven diagrams, in addition to the Use Case Model, for fully describing the solution for the system, so remember that you can save some work for later.

**Use Case relationships**

So far, actors, and Use Cases have been defined for the system, but there is need to associate each user with the system features they need to perform their jobs.

**Association notation**

A line connecting an actor to a Use Case represents an association. The association represents the fact that the actor communicates with the Use Case. In fact, in earlier versions of the UML spec, this was called a Communicates With relationship. This is the only relationship that exists between an actor and a Use Case. According to the UML spec, you may specify a directionality arrow on either end of the association line to denote the direction of the communication. Some associations are unidirectional (for example, the actor specifies information to the Use Case). Most associations are bidirectional (that is, the actor accesses the Use Case, and the Use Case provides functionality to the actor). For bidirectional associations, you may either place an arrowhead on both ends of the association line, or simply show no arrowheads at all. For simplification, most users tend to show no arrowheads at all. Most modeling tools provide the option to turn bidirectional arrows on or off. Just remember that the key is to identify which Use Cases the actors need to access. These connections will form the basis for the interfaces of the system and subsequent modeling efforts. The figure below illustrates this.

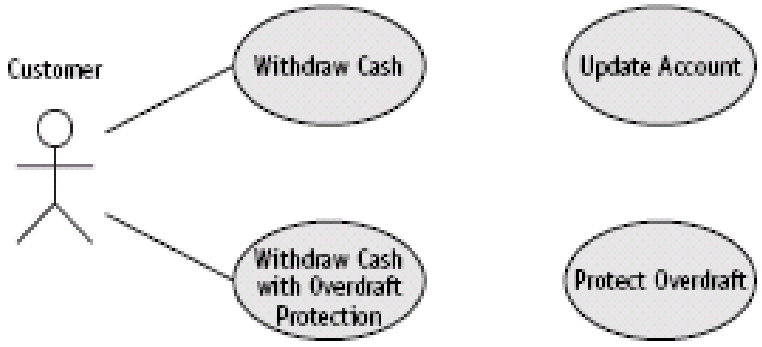



Figure showing Actor Use Case relationship

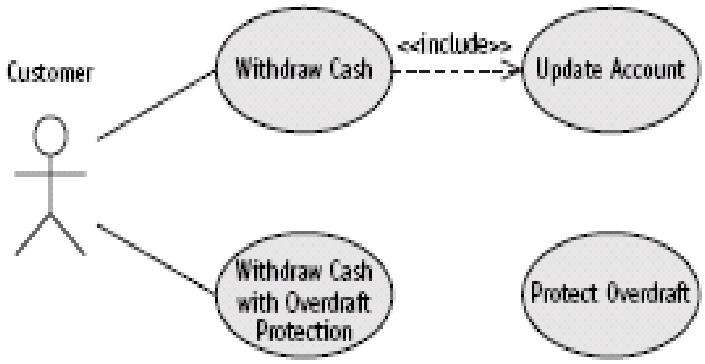
**Stereotype notation**

The stereotype notation is used throughout the UML, very commonly on Use Case dependencies, classes, and packages and other elements of the UML known as *classifiers*. The standard notation is to enclose the word in *guillemets* << >> (French quote marks), as in the **<<include>>** notation below. Stereotypes provide a means to extend the UML without modifying it. A stereotype functions as a qualifier on a model element, providing more information about the role of the element without dictating its implementation.

***<<include>> dependency notation***

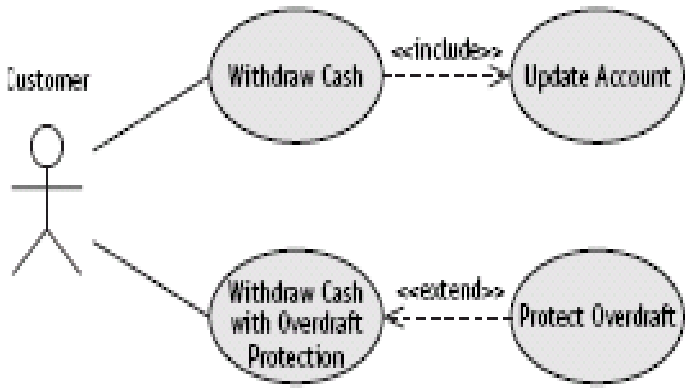
Sometimes one Use Case may need to ask for help from another Use Case. For example, Use Cases titled Deposit Money and Withdraw Money may not actually update a bank account. They may delegate the changes to an existing Use Case called Update Account so that changes are controlled through a single feature that guarantees that all changes are done correctly.

When one Use Case delegates to another, the dependency is drawn as a dashed arrow from the "using" Use Case to the "used" Use Case and labeled with the **<<include>>** stereotype notation, as shown in figure below. This conveys that executing the "using" (or calling) Use Case will include or incorporate the functionality of the "used" Use Case. If you have a programming background, you see right away the correlation with subroutine or function calls. Delegation may occur for one of two reasons. First, another Use Case may already exist to perform the task that is needed. Second, a number of Use Cases may need to perform the same task. Rather than write the same logic multiple times, the common task is isolated into its own Use Case and reused by, or included into, each Use Case that needs it.



**<<extend>> dependency notation**

The **<<extend>>** dependency stereotype says that one Use Case *might* need help from another Use Case. In contrast, the **<<include>>** dependency stereotype says that one Use Case will *always* call the other Use Case. Somewhere in the logic of the Use Case that needs the help is an *extension point,* a condition test that determines whether or not the call should be made. There is no such condition in an include dependency. The other contrast between the two dependency stereotypes is the direction of the dependency arrow. The **<<include>>** dependency arrow points from the main Use Case (the one currently executing) to the one that it needs help from. The **<<extend>>** dependency arrow points from the extension Use Case (the one providing the extra help) to the main Use Case that it is helping as shown in the figure below.




For example, the Withdraw Cash Use Case **<<includes>>** Update Account (the Withdraw Cash Use Case will always update the account). Likewise, the Protect Overdraft Use Case **<<extends>>** Withdraw Cash (the Protect Overdraft Use Case will sometimes be called by the Withdraw Cash Use Case).

**Generalization**

Inheritance is a key concept in object-oriented programming, and OO analysis and design. Inheritance tells us that one object has, at the time of its creation, access to all the properties of another class, besides its own class. Thus, the created object incorporates all those properties into its own definition. In layman's terms, we say things like, "A Ford Explorer is a car." A car is a well-defined general concept. When you create a Ford Explorer, rather than redefine all the car properties, you simply "inherit" or assimilate all the existing car properties, then override and/or add any new properties to complete the definition of your new Ford Explorer object.

The same idea, applied to actors and to Use Cases, is called *generalization,* and often goes by the nickname, an *"is a"* relationship. A Senior Bank Teller *is a* Bank Teller with additional authority and responsibilities. The "Withdraw Cash with Overdraft Protection" Use Case *is a* more extensive requirement than the "Withdraw Cash" Use Case. To model generalization, the UML uses a solid line with a hollow triangle. It looks a bit like an arrow, but be careful not to confuse the two. The triangle is always on the end near the item that is being inherited. In the examples earlier, the triangle would be near "Bank Teller" and "Withdraw Cash," as shown in figure below.

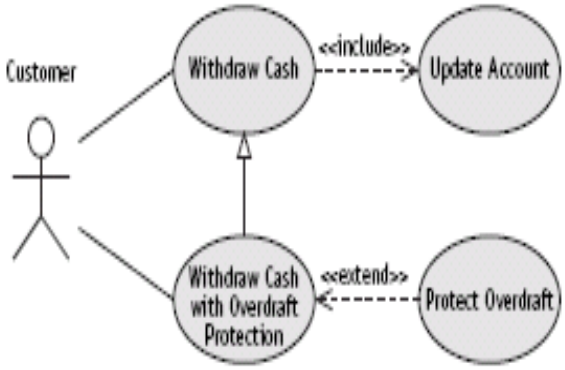



**Figure showing generalization**

**Building a Use Case Diagram**
**Case Study**

**Receiving:** The receiving clerks receive incoming shipments by matching purchase orders against the stock in the shipment. They inform the Accounts Payable department when the purchase order items have been received. The clients want the new system to handle the notification automatically.
**Stocking:** The products may come from cancelled orders, returned orders, or vendor shipments. The products are placed in the warehouse in predefined locations. The stock clerk looks up the correct location for the new products, places the products in that location, and updates the location inventory with the product quantity.
**Order Fulfillment:** Other staff members fill orders by locating the products required for the order. As they fill the order they update inventory to reflect the fact that they have taken the products. They also notify the Order Processing department that the order has been filled. The clients want the new system to handle the notification to Order Processing.
**Shipping:** When the orders are filled, they are then packed and prepared for shipping. The shipping people contact the shippers to arrange delivery. They then update inventory after they ship the product. They also notify the Order Processing department that the order has shipped. The clients want the new system to handle the notification to Order Processing.

***Step 1: Set the context of the target system***
Context always comes first. Context provides the frame of reference for the information you're evaluating. Context defines the placement of the system within the business, including the work processes, business plans and objectives, other systems, people and their job duties, and constraints imposed by external entities like government and contractual agreements.
According to the problem statement, the participants:

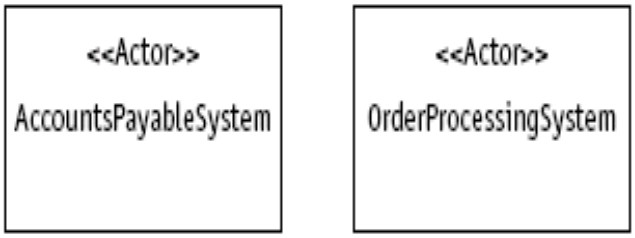
- ". . . inform the Accounts Payable department"
- ". . . notify the Order Processing department"
- ". . . contact the shippers"

The context places the system within the warehouse operations, working closely with Order Processing and Accounts Payable, and with shippers.

***Step 2: Identify the actors***

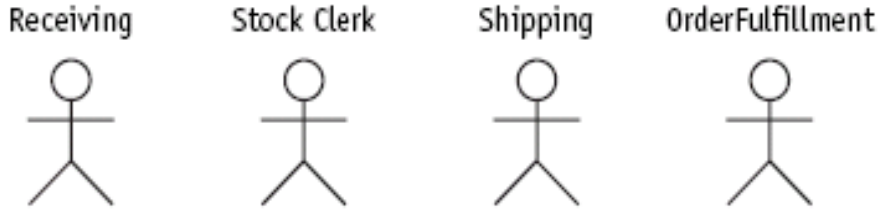
Find the people, systems, or devices that communicate with the system. The system-type actors are often easiest to spot as interfaces and external communication, such as notifications to the Accounts Payable and Order Processing systems. The other actors will be participants in the operation of the Inventory Control system. All these users will become your sources for finding and validating the required features of the system (that is, Use Cases).
The problem statement referred to two system-type actors, shown in the figure below:

- "They inform the *Accounts Payable department* when the purchase order items have been received." The Accounts Payable System must know when the company has incurred a liability for a shipment.
- "They also notify the *Order Processing department* that the order has been filled." "They also notify the *Order Processing department* that the order has shipped." The Order Processing System needs to keep the customer informed of the status of its shipment.




From the problem statement, you also find four human actors:

- "The receiving clerks receive incoming shipments by . . . ." People receive products into inventory.
- "The shipping people contact the shippers to . . . ." The people who ship the product, retain shippers, pack the product, and complete the shipping documents are referred to as *Shipping.*
- "Other staff members fill orders . . . ." The people responsible for filling orders, whether for samples, customer orders, wholesale, or retail, are referred to as *OrderFulfillment.*
- "The stock clerk looks up . . . ." The people responsible for putting the products into inventory are referred to as *Stock Clerk.*




***Step 3: Identify the Use Cases***
Find the features or functionality that the system must provide by asking these and similar questions:

- **What does the system produce for the actor?** This question helps identify work products that the system must support, known as the *critical outputs.*
- **What does the actor help the system do?** This question helps us know the input facilities that the system needs to support, known as the *critical inputs.*

- **What does the system help the actor(s) do?** This question helps identify the rules that must be applied when the actors use the system.

The Use Cases identified in the problem statement text include:

- **ReceiveProduct:** ". . . receive incoming shipments . . . ." The goal is to record products into inventory, regardless of source.
- **ShipOrder:** ". . . they ship the product." The goal is to record shipments and ensure that the products they contain have left the premises.
- **StockProduct:** "The products are placed in the warehouse in predefined locations." The goal is to record that products have been placed into the designated locations within the inventory.
- **FillOrder:** "Other staff members fill orders . . . ." The goal is to allocate specific inventoried products exclusively to satisfy an order.
- **LocateProduct:** "The stock clerk looks up the correct location . . . ." "Other staff members fill orders by locating . . . ." The goal is to identify the location within the facility in which a specific product resides.

***Step 4: Define the associations between actors and Use Cases***

Identify the actor(s) who need access to each Use Case/feature of the system. Each access relationship is a UML *association.* These associations are important because they tell you who the system *stakeholders* are (the people with a vested interest in the success of the system).

For example, will the person at the order desk be able to do his job if he can't see the status of an order? As a stakeholder, what does he have to say about how the Use Case should work?

- **An association between Receiving and ReceiveProduct.** "The receiving clerks receive incoming shipments . . . ."
- **An association between ReceiveProduct and AccountsPayableSystem.** "They inform the Accounts Payable department when the purchase order items have been received. The clients want the new system to handle the notification automatically."
- **An association between Shipping and ShipOrder.** "When the orders are filled, they are then packed and prepared for shipping. The shipping people contact the shippers to arrange delivery. They then update inventory once they ship the product."
- **An association between ShipOrder and OrderProcessingSystem.** "They also notify the Order Processing department that the order has shipped. The clients want the new system to handle the notification to Order Processing."
- **An association between StockClerk and Stock Product.** "The stock clerk looks up the correct location for the new products, places the products in that location, and updates the location inventory with the product quantity."
- **An association between FillOrder and OrderProcessingSystem.** "They also notify the Order Processing department that the order has been filled. The clients want the new system to handle the notification to Order Processing."
- **An association between OrderFulfillment and LocateProduct.** "Other staff members fill orders by locating the products required for the order."

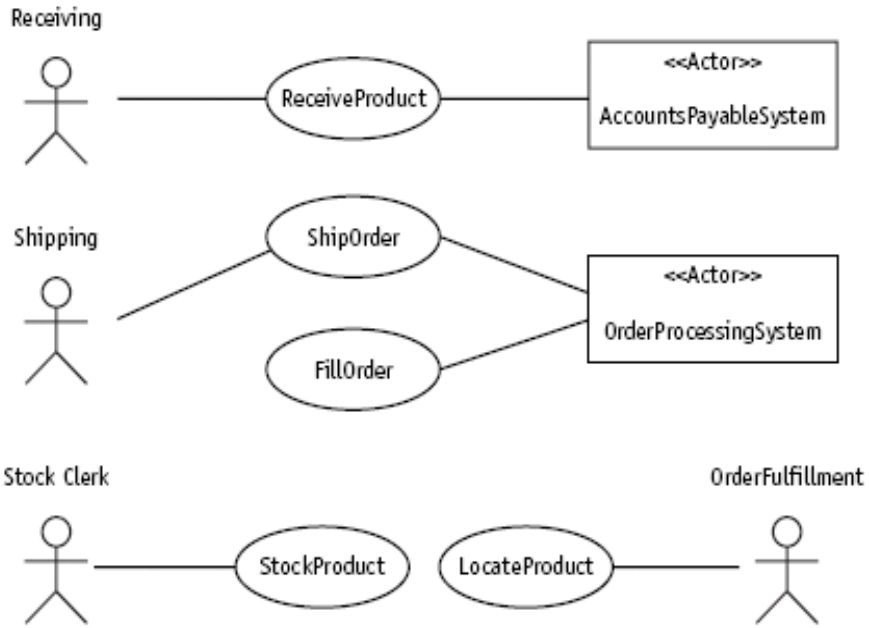



Figure showing associations from problem statement

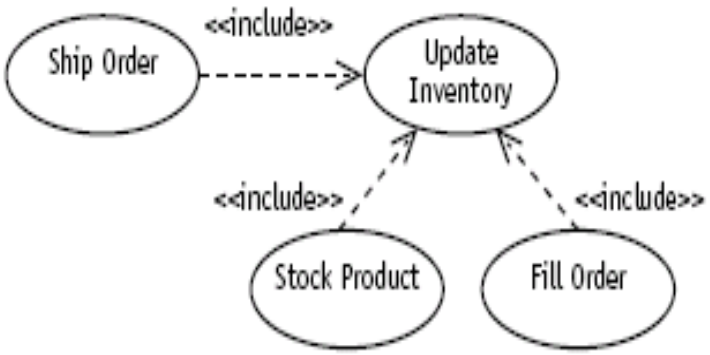
***Step 5: Evaluate the actors and Use Cases to find opportunities for refinement***

Rename, merge, and split actors and Use Cases as needed. When you build your diagrams based on interviews with users, it is easy to fall into the trap of replicating the current system. From your first draft of the descriptions of the actors and Use Cases, start asking critical questions, especially the simple but powerful question, "Why?"

For example, ask, "Why is this actor responsible for these particular duties?" or "Why do these tasks have to be done together, separately, in this order, or done at all?" A system rewrite or major revision provides a great opportunity to clean house and address a lot of the legacy problems that have accumulated over time.

***Step 6: Evaluate the Use Cases for <<include>> dependencies***

Apply the **<<include>>** dependency stereotype between Use Cases when one Use Case always calls on another Use Case to help it with a task that the calling Use Case cannot handle. The included Use Case may already exist or it may recur in a number of Use Cases and need to be isolated. For example, updating inventory is one of the requirements for ShipOrder, StockProduct, and FillOrder.




***Step 7: Evaluate the Use Cases for <<extend>> dependencies***

One Use Case may or may not use another Use Case depending upon a stated condition. When the condition is met, the call is made to the other Use Case. When the condition is not met, the call is not made.

***Step 8: Evaluate the actors and Use Cases for generalization***

Generalization is a tool for organizing similarities and differences among a set of objects (like actors or Use Cases) that share the same purpose.

The problem statement told us that, "The products may come from cancelled orders, returned orders, or vendor shipments." If the stocking rules are significantly different for the various types of incoming stock, you could use generalization on the StockProduct Use Case.

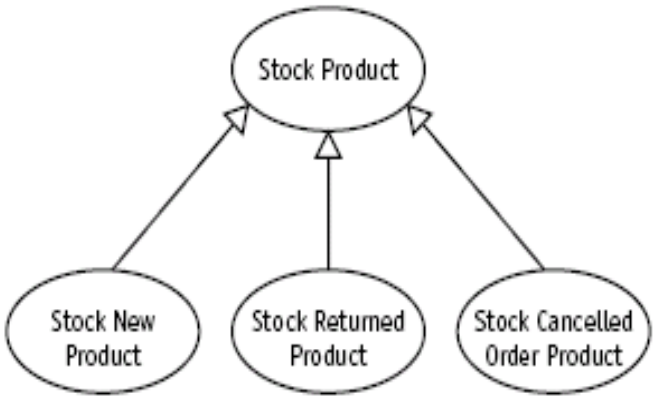



Figure showing generalization

The figure above shows that StockNewProduct inherits all the rules from StockProduct and then adds some variations unique to stocking new products. The same is true for StockReturnedProduct and StockCancelledOrderProduct.

**Building the Use Case Narrative**

Although the Use Case diagram provides a convenient view of the main features of a system, it is too concise to completely describe what users are expecting. So, as with most diagrams, it must

be supported by a narrative, a textual description that takes us to the next level of understanding. There are many ways to write Use Case descriptions. Typically, each methodology will have its own set of elements and preferences.

***Elements of a Use Case Narrative***

Describing a Use Case requires that you frame the context of the Use Case and describe the communication between the Use Case and the user, which could be an actor or another Use Case. With this in mind, most Use Case narratives include the following elements, or others very similar in meaning:

- Assumptions
- Pre-conditions
- Use Case initiation
- Process or dialog
- Use Case termination
- Post-conditions

***Assumptions***

Typically, developers think of assumptions as bad, something to be avoided. In order for the Use Case to work properly, certain conditions must be true within the system. In other words, assumptions describe a state of the system that must be true before you can use the Use Case. These conditions are *not* tested by the Use Case. For example, consider the tasks of performing authentication and authorization. A standard security check feature typically handles these functions. Each subsequent Use Case assumes that the user could not access the Use Case had he not made it past the security check. Consequently, you would rarely if ever include the security check in each Use Case.

If one Use Case can't work and should not even be accessed unless another Use Case has first done its job, this condition dictates the order of execution. The assumptions give you explicit clues about the sequence of execution for Use Cases (that is, the workflow).

***Pre-conditions***

Pre-conditions are easily confused with assumptions. Like assumptions, pre-conditions describe a state of the system that must be true before you can use the Use Case. But unlike assumptions, these conditions *are* tested by the Use Case before doing anything else. If the conditions are not true, the actor is refused entry. Most programmers have coded pre-conditions nearly every time they write a method or subroutine call that has parameters. When you write code, what are the first lines of code that you write in a function or method that has parameters? You validate the parameters. You test to make certain that the conditions are right to proceed with the rest of the code.

Failure in these tests would mean problems for the subsequent code, so the call is refused and turned back to the requester. You established and tested the pre-conditions for execution of your method or function.

These rules or pre-conditions need to be published along with the interface to your Use Case. For example, a typical interface can only tell the client to provide two integer values and a character string. It can't tell them the rules that say that the first integer must be a value between 1 and 10, the second must be an integer greater than 100, and the character string can only be 30 characters

in length. Without publishing these pre-conditions, anyone who wants to use your Use Case is forced into relying on trial and error to find the correct set of values.

***Use Case initiation***

A Use Case has to start somehow, but how? Some Use Cases start because an actor says, "Start." For example, you can select an option on a menu. The action tells the system to open the application. Time can also trigger a Use Case. Other Use Cases are implemented as objects themselves that watch for a point in time. A Use Case may be triggered by a system event like an error condition, a lost connection, or a signal from a device.

Use Case initiation provides a place to think through all the possible triggers that could launch the Use Case. This is critical when you start thinking about reusing Use Cases. If five actors and/or Use Cases plan on using the same Use Case, then you need to know how each user plans to kick it off. If each has different expectations, then you could be creating a problem. Multiple triggering mechanisms lead to high coupling and low independence. In other words, every time you change one of the triggers, you need to change the corresponding Use Case and make certain that you haven't created problems with the other triggering mechanisms. More triggers mean more complicated and costly maintenance.

***Dialog***

The *dialog* refers to a step-by-step description of the conversation between the Use Case (the system) and the user (an actor or another Use Case). Very often, it is helpful to model this sequence of events using an Activity diagram just as you might model a procedure for communication between two business units.

For example, you want to withdraw money, so you access the ATM at your local bank. The following dialog ensues:

*You get past the security check Use Case, and you're presented with a menu of options.*
*You choose "Withdraw."*

The system immediately asks you which account you want to withdraw the money from.

***Use Case termination***

Although there is usually only one triggering event to start a Use Case, there are often many ways to end one. You can pretty much count on some kind of normal termination where everything goes as planned and you get the result you anticipated. But things do go wrong. This could mean shutting down the Use Case with an error message, rolling back a transaction, or simply canceling. Each termination mechanism has to be addressed in the dialog.

The list of termination options is a bit redundant with the dialog, but just as was the case with pre-conditions, this redundancy provides some good checks and balances.

***Post-conditions***

Post-conditions describe a state of the system that must be true when the Use Case ends. You may never know what comes after the Use Case terminates, so you must guarantee that the system is in a stable state when it does end. In fact, some people use the term *guarantee* for just this reason. You guarantee certain things to be true when this Use Case completes its job. You might, for instance, guarantee to give the user a receipt at the end of the transaction, whether it succeeded or failed. You might promise to notify the user of the result of an attempted save to the database.

**Writing a Use Case Narrative for the Case Study**

To practice writing a narrative, you will use the Use Case Fill Order in the case study Use Case diagram presented in the figure below. Given the description in the next paragraph, you can draft a narrative for the FillOrder Use Case. Use the narrative elements discussed earlier as a guide for organizing the narrative.

FillOrder: This is basically your reason for being in business. Authorized personnel take Product from inventory according to the order specifications. They update the order and the inventory. If there are any items that can't be filled, they create a backorder.

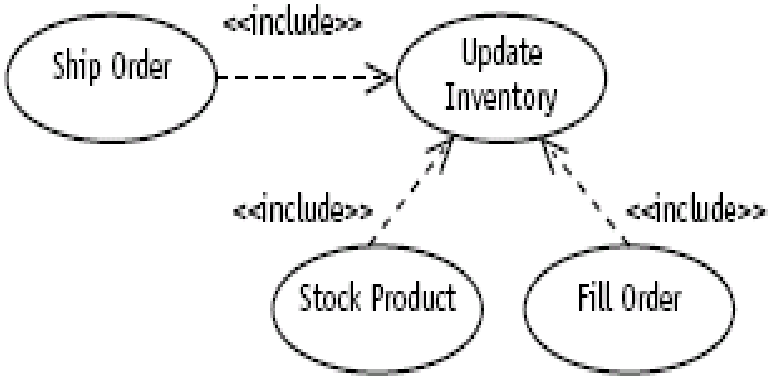



**Figure showing the FillOrder Use Case**

The narrative begins in the table below with four common audit fields to track the narrative document: the name of the Use Case, a unique number (in case you need to change the name), the author of the narrative, and the last time it was updated. You typically want to keep track of who is changing the document, what they have changed, and when they changed it, to make certain that everyone is aware of the latest revisions and to prevent confusion and unnecessary delays due to misunderstandings.

| Field Name | Field Description |
|---|---|
| Name | Fill Order |
| Number | 11 |
| Author | Tom Pender |
| Last update | 12/23/01 |

**Assumptions in the case study narrative**

The FillOrder description says that only "authorized personnel" will use this Use Case. You could check security in this Use Case. But that approach leads to redundancy and the resulting

high cost of maintenance. Instead you establish the assumption that security will be handled by another function, ValidateAccess. Furthermore, you'll trust that the security check was done correctly. So what does that tell you about the relationship between the FillOrder and ValidateAccess Use Cases? It tells you the precedence. ValidateAccess must precede FillOrder in the workflow.

| Field Name | Field Description |
|---|---|
| Assumptions | Valid user and has permission to use this feature |

***Pre-conditions in the case study narrative***

Next, you establish the pre-conditions, the conditions that you will test to be sure that it's okay to proceed with the Use Case dialog. The Fill Order description said that the personnel "take Product from inventory *according to the order specifications*." That implies that they need to tell the system the Order that they plan to fill. If the actor doesn't provide a valid order number, there isn't much you can do for him. Pre-conditions are always the first things tested in the dialog. The table below shows the pre conditions.

| Field Name | Field Description |
|---|---|
| Pre-conditions | Provide a valid order number |

**Use Case dialog in the case study narrative**

Next, you describe the dialog between the actor and the Use Case. You see each action required by the actor and each response from the system. Some responses require a choice based on the progress so far, so you see the decision statements included in the dialog. This way, you know why the system gave the response it did. Other decision points are the result of an action, like attempting to find the order using the Find Order Use Case. Below is the table showing the Case dialog in the Case Study.

| Field Name | Field Description |
|---|---|
| Use Case dialog | The system asks the user for an order number |
| | The user provides the order number |
| | The system asks for the order (from FindOrder Use Case). |
| | If the Order is not found, Error, stop |
| | Else: |
| | The system provides the order to the user |
| | The user chooses an item |
| | Until the user indicates that he is done or there are no unfilled item quantities greater than 0: |
| | The system asks for the location of the item and |
| | unfilled quantity (from the LocateProduct Use Case) |
| | If the item is found (available): |
| | The user indicates the quantity of the item filled |
| | If there are any unfilled item quantities greater than 0: |
| | Create a backorder (using the CreateBackorder Use Case) |

**Use Case termination in the case study narrative**

Use Cases are not good at showing concurrency and interrupts, so this is often the only place to identify things such as a cancel event and timeouts. You'll need to use the Activity diagram or even the Sequence diagram a bit later to flesh out the concurrency and interrupt requirements. The termination section also provides you with a list of actions to consider when writing the post-conditions.

NOTE
**No one diagram can show everything. This is why it is so important to understand the purpose and features of each diagram. The diagrams are like tools in a toolbox. You have to know which one to use for each task.**

Most business applications like FillOrder will let you cancel the session at specific points in the process. It is also usually wise to handle the condition where a user gets interrupted and leaves a session open. The timeout termination could watch for a specified period of inactivity before closing the transaction. And of course the user can actually complete the process of filling the order or simply indicate that he is done.

| Field Name | Field Description |
|---|---|
| Use Case termination | The user may cancel |
| | The Use Case may timeout |
| | The user can indicate that he is done |
| | The user can fill all items on the Order |

***Post-conditions in the case study narrative***

These conditions are especially important in that they reveal processing steps that may need to be added to the dialog to ensure that the system is stable when this Use Case is completed.

| Field Name | Field Description |
|---|---|
| Post-conditions | Normal termination:<br>The changes to the Order must be saved<br>(The backorder is handled by the CreateBackorder Use Case) |
| Cancel: | The Order must be saved unchanged<br>If a backorder was being created, it must be cancelled |

Note that the post-conditions include some items that go a bit beyond the scope of a Use Case, like saving the Order. A rule of thumb with Use Cases is that you include only what the user can see, and what can be inferred by what they can see. In this case, if the user gets a message indicating that the Order was updated, then you would include the messages in the Use Case dialog. You would not include the actual steps for updating the Order in the database. If your team feels that information is really needed, first make certain that users agree that your understanding, as documented in the narrative, was correct. Then you could take the Use Case description down a level to address the requirements you know have to be supported by the design. This rule is a little bit gray, but it comes with the territory.

# THE COPPERBELT UNIVERSITY

COMPUTER SCIENCE DEPARTMENT

**CS 432 – OBJECT ORIENTED SYSTEMS ANALYSIS AND DESIGN**

COMPILED BY DR NTALASHA

## Modeling the Static View: The Class Diagram

The Class diagram is by far the most used and best known of the object-oriented diagrams. It is the source for generating code and the target for reverse engineering code. Because the Class diagram is the primary source for code generation, the other diagrams tend to serve as tools of discovery that add to your knowledge about how to build the Class diagram. Use Cases identify the need for the objects as resources used by the system to achieve its goals. The Sequence and Collaboration diagrams are excellent tools for discovering object interactions and, by inference, defining interfaces. The Activity diagram is very good for discovering the behaviour implemented by objects and so helps to define the logic of operations on the objects.

## The Object Model

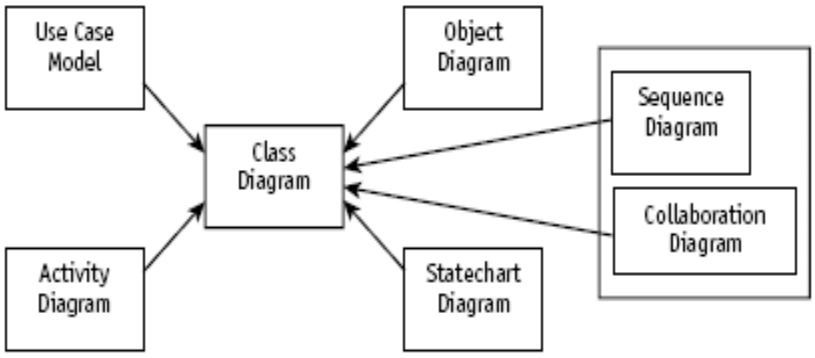
The phrase *object model* has been the source of some confusion. Object Model is often used as a synonym for Class diagram. In context, *object model* is used to mean the set of diagrams used to model objects, namely the Class and Object diagrams. The Class diagram is the more recognized and used of the two diagrams. The Object diagram is often implemented within the Class diagram, not as a separate diagram. In fact, the UML specification does not actually define the Object diagram. It is simply a Class diagram that contains only objects.

## The Class diagram

The Class diagram represents classes, their component parts, and the way in which classes of objects are related to one another. A class is a definition for a *type of object.* There are no real objects in a class, only descriptions of what a particular type of object looks like, what it can do, and what other objects it may be related to in some way. To document this information, the Class diagram includes attributes, operations, stereotypes, properties, associations, and inheritance.

- **Attributes** describe the appearance and knowledge of a class of objects.
- **Operations** define the behavior that a class of objects can manifest.
- **Stereotypes** help you understand this type of object in the context of other classes of objects with similar roles within the system's design.
- **Properties** provide a way to track the maintenance and status of the class definition.
- **Association** is just a formal term for a type of relationship that this type of object may participate in. Associations may come in many variations, including simple, aggregate and composite, qualified, and reflexive.
- **Inheritance** allows you to organize the class definitions to simplify and facilitate their implementation.

Together, these elements provide a rich set of tools for modeling business problems and software. However, the Class diagram is still limited in what it can show you. Generally speaking, it is a static view of the elements that make up the business or software. It's like a blueprint for a building or a piece of machinery. You can see the parts used to make it and how they are assembled, but you cannot see how the parts will behave when you set them into motion. This is why we need other diagrams to model behavior and interactions over time (that is, modeling the objects in motion). The figure below shows how all the other diagrams support the Class diagram.




Although other diagrams are necessary, remember that their primary purpose is to support the construction and testing of the Class diagram. Whenever another diagram reveals new or modified information about a class, the Class diagram must be updated to include the new information. If this new information is not passed on to the Class diagram, it will not be reflected in your code.

**The Object diagram**

The class defines the *rules;* the objects express the *facts.*
The class defines what *can be;* the object describes *what is.*
If the Class diagram says, "This is the way things should be," but the Object diagram graphically demonstrates that "it just ain't so," then you have a very specific problem to track down. The reverse is true, too. The Object diagram can confirm that everything is working as it should.

**Elements of the Class Definition**

- The *class symbol* is comprised of three *compartments* (rectangular spaces) that contain distinct information needed to describe the properties of a single type of object.
- The *name compartment* uniquely defines a class (a type of object) within a package. Consequently, classes may have the same name if they reside in different packages.
- The *attribute compartment* contains all the data definitions.
- The *operations compartment* contains a definition for each behaviour supported by this type of object.

**Modeling an Attribute**

An attribute describes a piece of information that an object owns or knows about itself. To use that information, you must assign a name and then specify the kind of information, or data type. Data types may be primitive data types supplied by a language, or abstract data types (types defined by the developer). In addition, each attribute may have rules constraining the values assigned to it. Often a default value helps to ensure that the attribute always contains valid, meaningful data.

**Attribute visibility**

Each attribute definition must also specify what other objects are allowed to see the attribute - that is its *visibility.* Visibility is defined as follows:

- Public (+) visibility allows access to objects of all other classes.
- Private (-) visibility limits access to within the class itself. For example, only operations of the class have access to a private attribute.
- Protected (#) visibility allows access by subclasses. In the case of *generalizations* (inheritance), subclasses must have access to the attributes and operations of the superclass or they cannot be inherited.
- Package (~) visibility allows access to other objects in the same package.

Given these requirements, the following notation is a common way of defining an attribute:

**visibility / attribute name : data type = default value {constraints}**

- **Visibility (+, -, #, ~):** Required before code generation. The programming language will typically specify the valid options. The minus sign represents the visibility "private" meaning only members of the class that defines the attribute may see the attribute.
- **Slash (/):** The derived attribute indicator is optional. Derived values may be computed or figured out using other data and a set of rules or formulae. Consequently, there are more design decisions that need to be addressed regarding the handling of this data. Often this flag is used as a placeholder until the design decisions resolve the handling of the data.
- **Attribute name:** Required. Must be unique within the class.
- **Data type:** Required. This is a big subject. During analysis, the data type should reflect how the client sees the data. You could think of this as the external view. During design, the data type will need to represent the programming language data type for the environment in which the class will be coded. These two pieces of information can give the programmer some very specific insights for the coding of **get** and **set** methods to support access to the attribute value.
- **Assignment operator and default value:** Optional. Default values serve two valuable purposes. First, default values can provide significant ease-of-use improvements for the client. Second and more importantly, they protect the integrity of the system from being corrupted by missing or invalid values.

**Modeling an Operation**

Objects have behaviours, things they can do and things that can be done to them. These behaviours are modeled as operations. By way of clarification, the UML distinguishes between operation and method, whereas many people use them interchangeably. In the UML, an operation is the declaration of the signature or interface, the minimum information required to invoke the behavior on an object. A method is the implementation of an operation and must conform to the signature of the operation that it implements.

**Elements of an operation specification**

Operations require a name, arguments, and sometimes a return. Arguments, or input parameters, are simply attributes, so they are specified using the attribute notation (name, data type, constraints, and default), although it is very common to use the abbreviated form of name and data type only.

Note that if you use constraints on an argument, you are constraining the input value, not the value of the attribute as it resides in the source object. The value in the source object was constrained in the attribute definition within the class.

The return is an attribute data type. You can specify the visibility of the operation:
private (-) limits access to objects of the same class, public (+) allows access by any object, protected (#) limits access to objects of subclasses within the inheritance hierarchy (and sometimes the same package), and package (~) limits access to objects within the same package. Given these requirements, the following notation is used to define an operation:

**visibility operationName ( argname : data type {constraints}, ...) : return data type {constraints}**

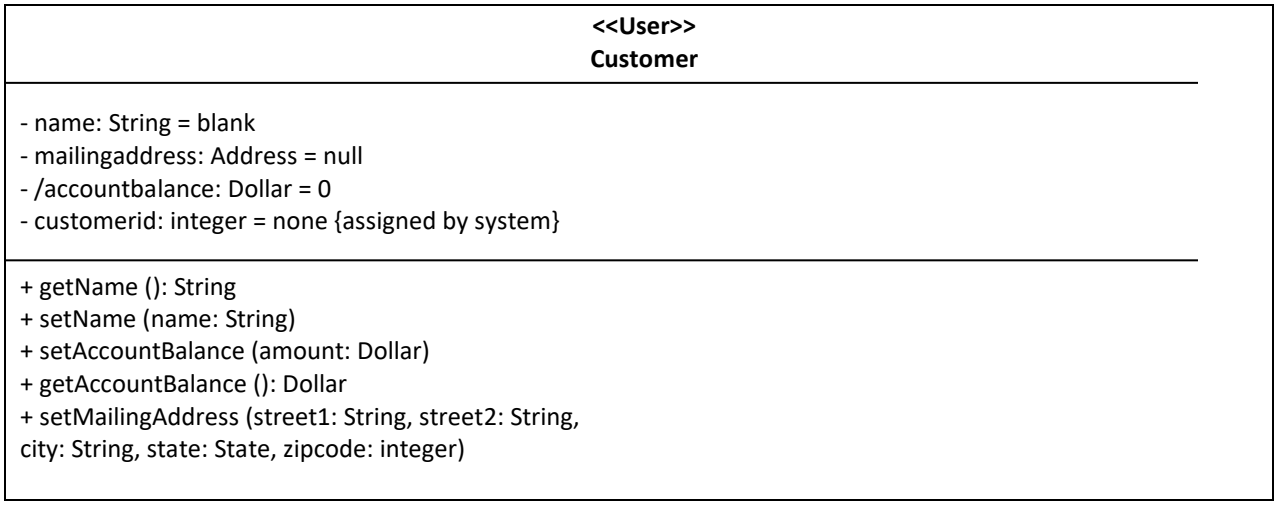
- **Operation name:** Required. Does not have to be unique, but the combination of name and parameters does need to be unique within a class.
- **Arguments/parameters:** Any number of arguments is allowed. Each argument requires an identifier and a data type. Constraints may be used to define the valid set of values that may be passed in the argument. But constraints are not supported in many tools and will not be reflected in the code for the operation, at least not at this point.
- **Return data type:** Required for a return value, but return values are optional. The UML only allows for the type, not the name, which is consistent with most programming languages. There may only be one return data type, which again is consistent with most programming languages.
- **Visibility (+, -, #, ~):** Required before code generation. The visibility values are defined by the programming language, but typically include public (+), private (-), protected (#), and package (~).
- **Class level operation (underlined operation declaration):** Optional. Denoted as an operation accessible at the class level; requires an instance (object) reference.
- **Argument name:** Required for each parameter, but parameters are optional. Any number of arguments is allowed.
- **Argument data type:** Required for each parameter, but parameters are optional.
- **Constraints:** Optional. In general, constraints express rules that must be enforced in the execution of the operation. In the case of parameters, they express criteria that the values

must satisfy before they may be used by the operation. You can think of them as operation level pre-conditions.

**Modeling the Class Compartments**

Now you need to put all this together in a class symbol. The class notation consists of the three compartments mentioned earlier. You've just seen the contents of the second and third compartments for attributes and operations, respectively. The first compartment - the name compartment—gives the class its identity.

The figure below shows a UML class symbol with all three compartments and all the elements needed to define a class with UML notation. The text addresses each compartment of the class.




**Name compartment**

In the figure above, the name compartment occupies the top section of the class box. The name compartment holds the class name, an optional stereotype, and optional properties. The name is located in the center of the compartment. The stereotype (<< >>) may be used to limit the role of the class in the model and is placed at the top of the compartment. Common examples of class stereotypes include **<<Factory>>**, based on the Factory design pattern, and **<<Interface>>**, for Java interfaces or for user interfaces.

Properties use the constraint notation { } and are placed in the bottom-right corner of the compartment. Properties are basically constraints used to clarify the intent in defining the model element. Properties can be used to document the status of a class under development or for designations such as *abstract* and *concrete.*

**Attribute compartment**

The attribute compartment occupies the middle section of the class box. The attribute compartment simply lists the attribute specifications for the class using the notation presented earlier in "Modeling an Attribute." The order of the attributes does not matter.

**Operation compartment**

The operations compartment occupies the bottom section of the class box. Operations are simply listed in the operation compartment using the notation presented in "Modeling an Operation."

The order does not matter. The operation compartment is placed below the name compartment, and below the attribute compartment when all compartments are visible.

**The Class Diagram: Associations**

Associations between objects are similar to associations between people. In order for me to work with you, I need to communicate with you. This requires that I have some way to contact you, such as a phone number or an e-mail address. Further, it is often necessary to identify why we are associated in order to clarify why we do and do not participate in certain kinds of communication. For example, if we are associated because you are a programmer and I am a database administrator, we probably will not discuss employee benefits as part of our duties.
There would also probably be some limitations placed on our interactions:

- We would want to limit the number of participants in the relationship to ensure efficiency.
- We would want to check the qualifications of the participants to ensure we have the right participants.
- We would want to define the roles of the participants so that everyone knows how to behave.

All these requirements apply equally to objects. The UML provides notations to address them all.

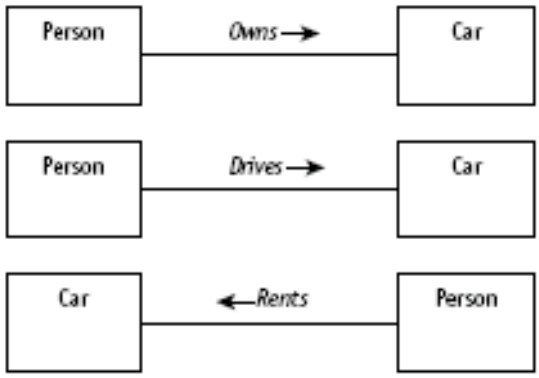
**Modeling Basic Association Notations**
The following notations appear on almost every association you will model. Most of these elements are similar to those you find in data modeling or database design. In fact, most of the concepts come from these fields. The concepts worked well in data modeling, so they were simply brought forward into object modeling as a form of "best practices."

**Association name**
The purpose of the association can be expressed in a *name*, a verb or verb phrase that describes how objects of one type (class) relate to objects of another type (class). For example, a person owns a car, a person drives a car, and a person rents a car. Even though the participants are the same in each association, the purpose of each association is unique, and as such they imply different rules and interactions.
To draw the UML associations for these three examples, you need to start with four basic elements.

- The *participating classes,* Person and Car.
- The *association,* represented by a line between the two classes.
- The *name* of the association, represented by a verb or verb phrase on the association line. Don't worry about the exact position of the name. As long as the name appears somewhere in the middle of the line, you're okay.
- The *direction* to read the name (indicating the direction is optional).

Person owns Car and Person drives Car. Note that if these two statements are true, then the reverse would be equally true—Car is owned by Person and Car is driven by Person. Associations may be read in both directions as long as you remember to reverse the meaning of the association name from active to passive.

The association name would not make sense if you read it in the typical left to right fashion—Car rents Person. This is a case where the direction indicator is particularly appropriate, even required, to make sense of the association by reversing the normal reading order so that it reads from right to left—Person rents Car.




**Association multiplicity**

The UML allows you to handle some other important questions about associations: "How many Cars may a Person own?" "How many can they rent?" "How many people can drive a given Car?" Associations define the rules for *how* objects in each class may be related. So how do you specify exactly *how many* objects may participate in the relationship?

*Multiplicity* is the UML term for the rule that defines the number of participating objects. A multiplicity value *must* be assigned to each of the participating classes in an association.

Multiplicity is expressed in a couple of ways. The most common is a range defining the minimum number of objects allowed and the maximum number of objects allowed in the format

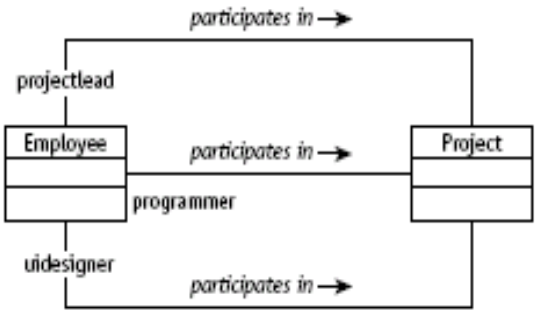
**Minimum . . Maximum**

You must use integer values for the minimum and maximum. But, as you have probably found in your own applications, sometimes you don't know the upper limit or there is no actual upper limit. The UML suggests the use of an asterisk to mean *no upper limit.* Used by itself, the asterisk can also mean the minimum is zero *and* there is no upper limit, or *zero or more.*

**Association roles**

Sometimes the association name is a bit hard to determine. For example, what English word could you use for the association name between parents and children? The UML provides an alternative that may be used in place of the name or along with it to help make the reason for the association as clear as possible. This alternative is called a *role* because it describes *how an object participates* in the association.

For example, many employees contribute to a project. But you know from experience that they participate in different ways. The figure below shows how you can draw multiple associations and label them to differentiate the types of participation. Each role is placed at the end of the association next to the type of object that plays the role. You may use them on one, both, or neither end of each association.




There is one other thing worth noting about roles and names. Role names generate code. Association names *do not* generate code. The role name can be used to name the attribute that holds the reference to the object that plays the role. In the figure above, the **Project** object could have an attribute named **programmer** that holds a reference to an **Employee** object that plays the role of programmer, and another attribute called **projectlead** that holds reference to another **Employee** object that plays the role of project lead.

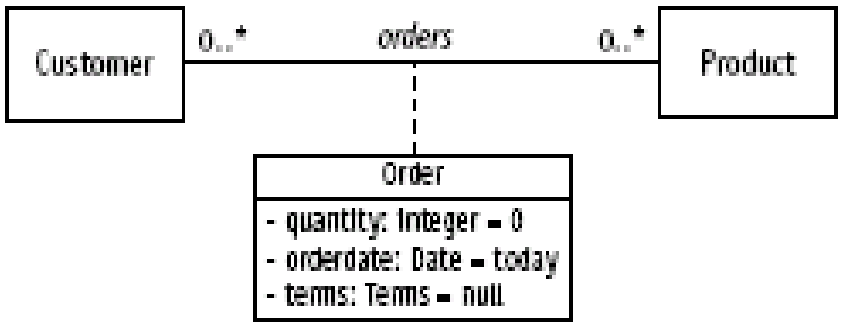
**Association constraints**

Constraints appear throughout the UML notation. Constraints fulfill much the same function for associations.


**Modeling Extended Association Notations**

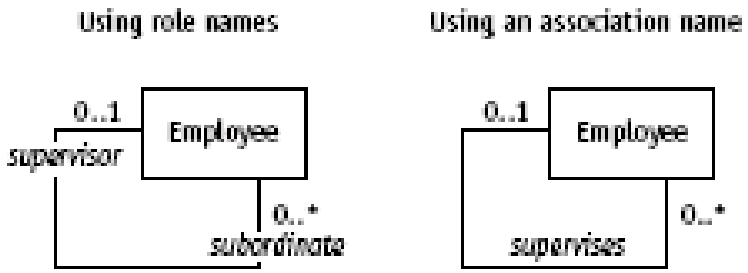
**Association class**

An association class encapsulates information about an association. An association class encapsulates information about an association. In the figure below you know that Customers order Products. But when customers order products there is usually more that you need to know, like when did they order the products? How many did they order? What were the terms of the sale? All the answers to these questions are simply data. All data in an object-oriented system must be contained in (encapsulated in) an object. There must be a class to define each type of object. So, define all this data in a class. Then to show that the data describes the association, attach the new class to the association with a *dashed line.*



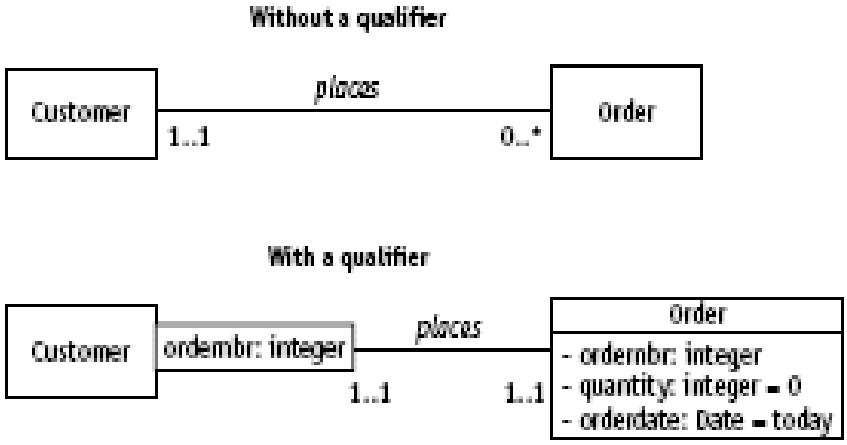

**Reflexive association**

Reflexive association is a fancy expression that says objects in the same class can be related to one another. The entire association notation you've learned so far remains exactly the same, except that both ends of the association line point to the same class. This is where the reflexive association gets its name. The association line leaves a class and reflects back onto the same class.
Both examples in the figure are equivalent expressions. The only difference is that one uses roles and the other uses an association name.




**Qualified association**

Qualified associations provide approximately the same functionality as indexes, but the notation has a bit of a twist. To indicate that a customer can look up an order using the order's ordernumber attribute, place the ordernumber attribute name and data type in a rectangular box on the Customer end of the association. The rest of the association notation remains intact but is pushed out to the edge of the rectangle.



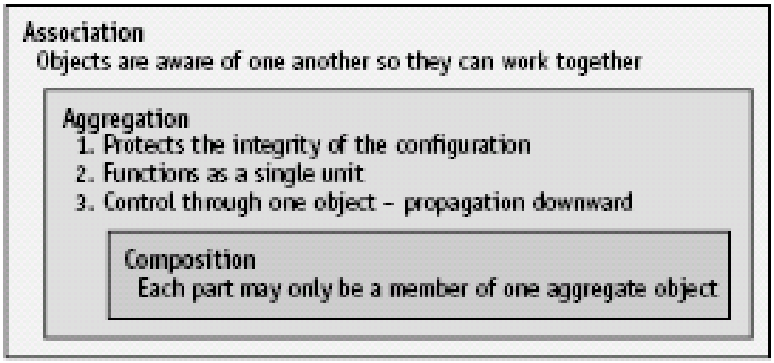

**The Class Diagram: Aggregation and Generalization**

An association describes a set of rules regarding how objects may be related to one another. But associations can be a bit more restrictive. There are two common subtypes of association, called aggregation and composition.

**Modeling Aggregation and Composition**

The figure below outlines the relationships among the concepts of association, aggregation, and composition.

- Every aggregation relationship *is a type of association.* So every aggregation relationship has all the properties of an association relationship, plus some rules of its own.
- Every composition relationship *is a form of aggregation.* So every composition relationship has all the properties of an aggregation, plus some rules of its own.




**Elements of aggregation**

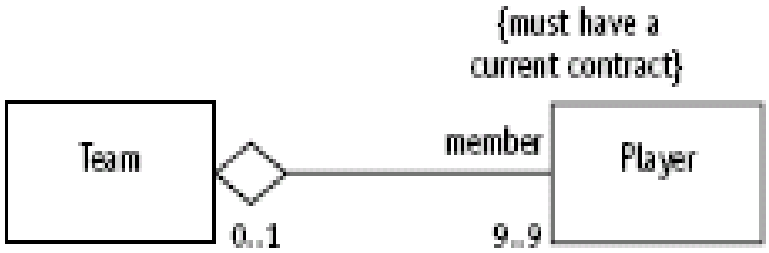
*Aggregation* is a special type of association used to indicate that the participating objects are not just independent objects that know about each other. Instead, they are *assembled* or *configured* together to create a new, more complex object. For example, a number of different parts are

assembled to create a car, a boat, or a plane. You could even create a logical assembly like a team where the parts are not physically connected to one another but they still operate as a unit.

To model aggregation on a Class diagram:

- Draw an association (a line) between the class that represents the member and the class that represents the assembly or aggregation. In the figure below, that would mean a line between the Team class and the Player class.
- Draw a *diamond* on the end of the association that is attached to the assembly or aggregate class. In the same figure, the diamond is next to the Team class that represents a group of players.
- Assign the appropriate multiplicities to *each end* of the association, and add any roles and/or constraints that may be needed to define the rules for the relationship.

In the figure, a Player may be a member of no more than one Team, but a Player does not have to be on a Team all the time (0..1). The Team is always comprised of exactly nine Players (9..9 or just 9). A Player is considered a *member* (role name) of a Team. Every Player is constrained by the fact that she must have a current contract in order to be a member of a Team.

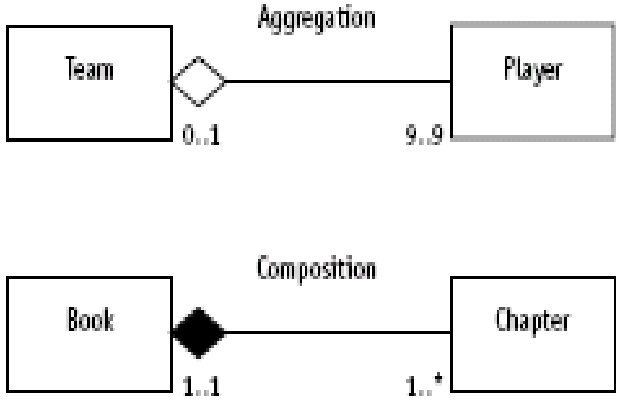



Aggregation describes a group of objects in a way that changes how you interact with them. The concept is aimed at protecting the integrity of a configuration of objects in two specific ways.
First, aggregation defines a single point of control in the object that represents the assembly. This ensures that no matter what others might want to do to the members of the assembly, the control object has the final word on whether the actions are allowed. This assignment of control may be at many levels within the aggregation hierarchy. For example, an engine might be the controller of its parts, but the car is the controller of the engine.
Second, when an instruction is given that might effect the entire collection of objects, the control object dictates how the members will respond. So for all intentions, the assembly appears to function like a single object. When gas pedal is pushed, telling the car that it should accelerate, the entire car assembly (with its thousands of parts) accelerates, not just the gas pedal.

**Elements of composition**

*Composition* is used for aggregations where the life span of the part depends on the life span of the aggregate. The aggregate has control over the creation and destruction of the part. In other words, the member object cannot exist apart from the aggregation. Draw this stronger form of aggregation simply by making the aggregation diamond solid (black).
In the figure below, the team example uses aggregation, the hollow diamond. Players are assembled into a team. But if the Team is disbanded, the players live on (depending of course on

how well they performed). The Book example uses composition, the solid diamond. A Book is composed of Chapters. The Chapters would not continue to exist elsewhere on their own. They would cease to exist along with the Book.

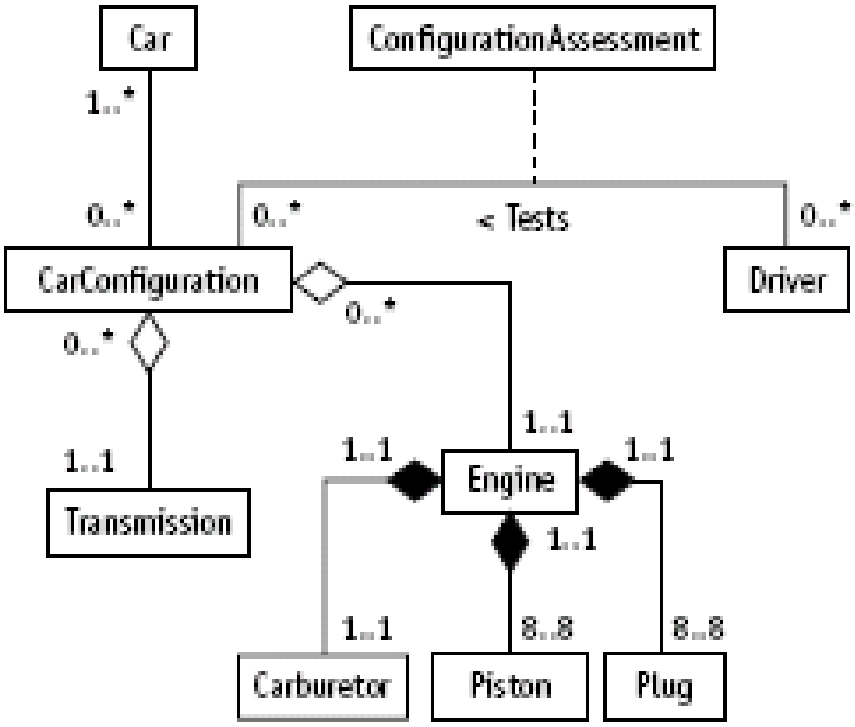



Note how the multiplicity provides some clues on the distinction between aggregation and composition. On the Team example, each Player may or may not be a member of a Team (0..1). This tells me that a Player may exist independent from the Team.

The Book example says that a Chapter must be associated with one and only one Book (1..1). This tells me that a Chapter cannot exist independent of the Book, so it must be a composition relationship.

**Creating aggregation and composition relationships**

**Problem statement:** "Our Company maintains a group of race cars. Our cars use some of our new 8-cylinder engines and new transmissions. Once the engines are assembled, the pistons, carburetor, and plugs cannot be swapped between engines due to changes caused by the high temperatures.
"We want to keep records of the performance achieved by each engine in each car and each transmission in combination with each engine. Our drivers evaluate each car to give us their assessment of the handling. We need a system to track the configurations and the drivers' assessments."
The figure below shows that each car can be configured with different engine and transmission combinations. The configuration is built using an engine and a transmission (aggregation). The drivers test the car configurations. The association class Configuration Assessment contains the details of the drivers' evaluations for each configuration.




The engine is composed of the carburetor, pistons, and plugs (along with other unnamed parts). However, these parts become permanent parts of the engine once installed, so they are modeled with composition instead of aggregation.

# THE COPPERBELT UNIVERSITY

COMPUTER SCIENCE DEPARTMENT

**CS 432 – OBJECT ORIENTED SYSTEMS ANALYSIS AND DESIGN**

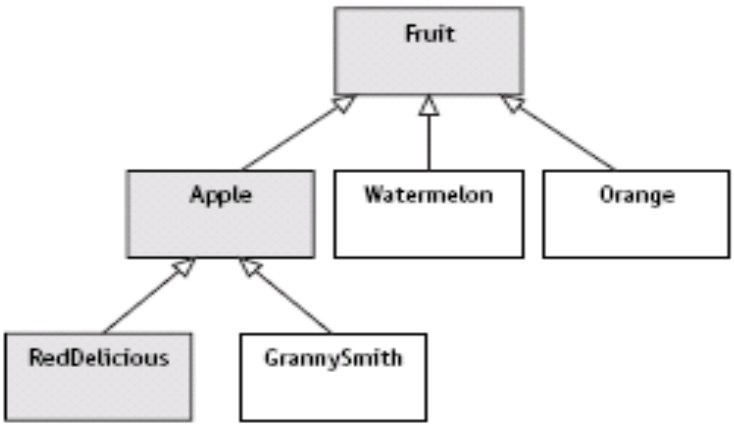
COMPILED BY DR NTALASHA

## Modeling Generalization

*Generalization* is the process of organizing the properties of a set of objects that share the same purpose. People use this process routinely to organize large amounts of information. Walk through a grocery store and you will find foods located in areas of the store depending upon their properties. Dry goods are located in one area, fruits and vegetables in another, meat in yet another. All these items are foods, but they are different kinds of foods or types of foods. Words such as *kind of* or *type of* are often used to describe a generalization relationship between classes (for example, an apple is a type of fruit that is in turn a kind of food and so on).

You might also hear this type of relationship called *inheritance.* Many times the terms *generalization* and *inheritance* are used synonymously. If an apple is a kind of fruit, then it *inherits* all the properties of fruit. Likewise, an apple is a *specialization* of fruit because it inherits all the generalized properties of fruit and adds some unique properties that only apply to apples. In the reverse, the concept fruit is a *generalization* of the facts that are true for watermelons, apples, peaches, and all types of objects in the group.

A generalization is *not* an association. In fact, association and generalization are treated as separate model elements in the UML metamodel. Associations define the rules for how *objects* may relate to one another. Generalization relates *classes* together where each class contains a subset of the elements needed to define a type of object. Instantiating all the element subsets from each of the classes in a single inheritance path of the generalization results in one object. In the Fruit illustration in the figure below, to create a Red Delicious Apple, we need to combine the RedDelicious class, the Apple class, and the Fruit class to get all the attributes and operations that define a Red Delicious Apple. From the combined class create (instantiate) an object of type RedDelicious (apple).




**Elements of generalization**

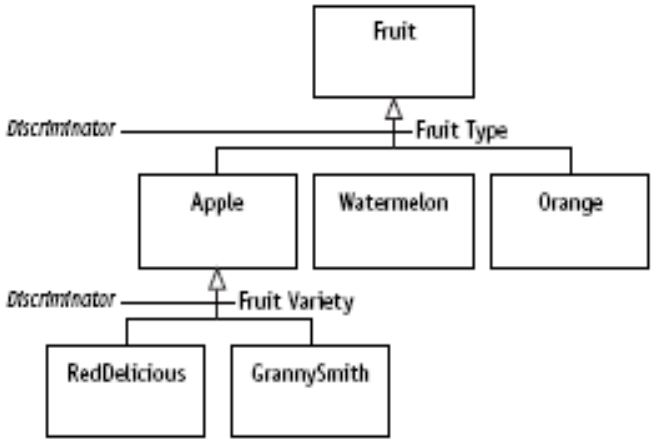
Because the generalization (also called an *inheritance*) relationship is *not* a form of association, there is no need for multiplicity, roles, and constraints. These elements are simply irrelevant.
To draw a generalization relationship, we first need to define *superclass, subclass,*abstract class, concrete class, and *discriminator.* A superclass is a class that contains some combination of attributes, operations, and associations that are common to two or more types of objects that share the same purpose. Fruit and Apple are examples of superclasses.
The term *superclass* reflects the concept of superset. The superset or superclass in this case contains the traits that are common to every object in the set.

A *subclass* is a class that contains some combination of attributes, operations, and associations that are unique to a type of object that is partially defined by a superclass.
The term s*ubclass* reflects the concept of subset. The subset, or subclass, contains a unique set of properties for only certain objects within the set.

An *abstract* class is a class that cannot create objects (cannot be instantiated). Any superclass that defines at least one operation that does not have a method is said to be abstract, or lacking a complete definition. Only a superclass can be abstract.

A *concrete* class is a class that has a method for every operation, so it can create objects. The methods may be defined in the class or inherited from a superclass. All classes at the bottom of a generalization hierarchy *must* be concrete. Any superclass *may* be concrete.

A *discriminator* is an attribute or rule that describes how to identify the set of subclasses for a superclass. If you wanted to organize the information about types of cars, you could discriminate based on price range, manufacturer, engine size, fuel type, usage, or any number of other criteria. The discriminator choosen depends on the problem being solved. The figure below illustrates this concept.




The composition of a class reveals the possible discriminating properties. Classes define properties of objects such as attributes, operations, and association. These are the first three possible discriminating properties. If objects of the class share the same attributes, like age and address, they might be in the same subgroup. However, objects might have the same attribute (like age), but the values allowed for age in some of the objects are different from those allowed in others. For example, every Person has an age value assigned. However, minors would have age values less than 21 (in some states) and adults would have ages greater than 20.
The same concept applies to operations. Objects might have the same operation, that is, the same interface, like "accelerate." But different objects might implement that interface in very different ways. A car accelerates very differently from a rocket. Even different cars accelerate using different combinations of parts and fuels.

In summary, there are at least five objective criteria used to discriminate between objects within the same class (superclass):

- Attribute type
- Attribute values allowed
- Operation (interface)
- Method (implementation)
- Associations

# THE COPPERBELT UNIVERSITY

COMPUTER SCIENCE DEPARTMENT

**CS 432 – OBJECT ORIENTED SYSTEMS ANALYSIS AND DESIGN**

COMPILED BY DR NTALASHA

## Modeling the Functional View: The Activity Diagram

The Activity diagram is part of the Functional view because it describes logical processes, or functions, implemented using code. Each process describes a sequence of tasks and the decisions that govern when and how they are done. It's a must to understand these processes in order to write correct code.
Functional modeling has acquired a poor reputation with the onset of object-oriented (OO) modeling. But both functional modeling and data modeling provide valuable insight into software development. OO development methods simply bring the two concepts together. Functional modeling is still a very basic part of any application design.

### Introducing the Activity Diagram

In the past, you used flowcharts, a simple technique with a wide range of applications. The
UML offers an enhanced version of flowcharts in the form of the Activity diagram, the focus of this session.
Where might you use the Activity diagram? There are at least three places in the UML where an Activity diagram provides valuable insight: workflow, Use Cases, and operations.

### Modeling workflow and Use Cases

When modeling a Use Case, you're attempting to understand the goal that the system must achieve in order to be successful. Use the Activity diagram to follow the user through a procedure, noting the decisions made and tasks performed at each step. The procedure may incorporate many Use Cases (workflow), a single Use Case, or only part of a Use Case.

A workflow-level Activity diagram represents the order in which a set of Use Cases is executed. An Activity diagram for one Use Case would explain how the actor interacts with the system to accomplish the goal of the Use Case, including rules, information exchanged, decisions made, and work products created.
Modeling the user's work this way does not bind you to that particular version of the process. Remember that for each goal (Use Case), there may be any number of valid processes. But creating such a model will likely reveal the essential elements of the process in a way that is familiar to the users. The new presentation then facilitates the interview to clarify the tasks and the reasons behind them.

### Defining methods

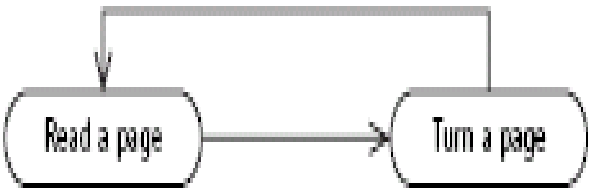
The Activity diagram can also be used to model the implementation of complex methods. When defining the implementation for an operation, you need to model the sequence of data

manipulation, looping, and decision logic. Modeling complex functions can prevent problems later when you write the code by revealing all the requirements explicitly in the diagram. Models make most, if not all, of your assumptions visible and consequently easier
to review and correct.

That last statement is worth emphasizing. It is very tempting to shortcut the modeling process, but the true value of modeling is in revealing what you know so that it can be challenged and verified. Making assumptions sabotages this very valuable benefit of modeling.
The Activity diagram contains all the logical constructs you find in most programming languages. In fact, it can translate quite well into pseudo-code or even native code. The word *operation* in the UML refers to the declaration portion of a behavior defined by a class. This typically means the name, parameters, and possibly the return type. This is also often called the *interface*, the information you use to tell the object what you want it to do.
In contrast, the word *method* is used to refer to the implementation of an operation, the code that is executed when you invoke the interface. The Activity diagram may be used to design the requirements for a method. Because the UML does not provide a place to model, the methods in a Class diagram, or anywhere else, the Activity diagram can fill in this missing piece.
Not everyone has accepted or uses the definitions for operation and method as they are used in the UML. To be sure, not every operation is complicated enough to warrant drawing an Activity diagram. The point is that the Activity diagram is well suited to the task when it is needed.

**Activity Diagram Notation**

**Activities and transitions**
An *activity* is a step in a process where some work is getting done. It can be a calculation, finding some data, manipulating information, or verifying data. The activity is represented by a rounded rectangle containing freeform text. An *Activity diagram* is a series of activities linked by *transitions*, arrows connecting each activity. Typically, the transition takes place because the activity is completed. For example, you're currently in the activity "reading a page." When you finish this activity, you switch to the activity "turning page." When you are done turning the page . . . well, you get the idea. The figure below shows this idea graphically.




This notation starts to show the overlap between the Activity diagram and the Statechart diagram. In fact, the Activity diagram is a subset of the Statechart diagram. Each Activity is an *action state* where the object is busy doing something (as opposed to waiting). Each transition is a change in state, a change from one activity or active state to the next. So as you learn the Activity diagram, you are well on your way toward understanding the Statechart diagram as well.

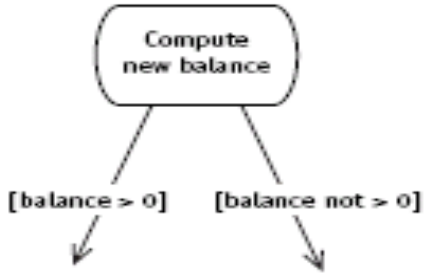
**Guard condition**

Sometimes the transition should only be used when certain things have happened. A *guard condition* can be assigned to a transition to restrict use of the transition. Place the condition within square brackets somewhere near the transition arrow. The condition must test true before you may follow the associated transition to the next activity. The Activity diagram segment in the Figure below tells you that you can't leave the table when you've finished your dinner unless you have finished your vegetables.


**Decisions**

The Activity diagram diamond is a decision icon, just as it is in flowcharts. In either diagram, one arrow exits the diamond for each possible value of the tested condition. The decision may be as simple as a true/false test. Each option is identified using a guard condition. Each guard condition must be mutually exclusive so that only one option is possible at any decision point. The guard is placed on the transition that shows the direction that the logic follows if that condition is true. If you write code, then you have probably used a case statement to handle this same type of problem.

Because every choice at a decision point is modeled with a guard condition, it is possible to use the conditional logic on transitions leaving an activity as well. All the information needed to make the choice is provided by the activity. To show the choices resulting from an activity, simply model the transitions exiting the activity, each with a different guard condition. The figure below illustrates this.

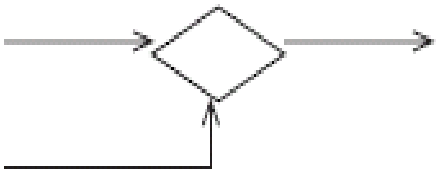



**Merge point**

The diamond icon is also used to model a *merge point,* the place where two alternative paths come together and continue as one. The two paths in this case are mutually exclusive routes.

You can also think of the merge point as a labor-saving device. The alternative would be to model the same sequence of steps for each of the paths that share them.

The figure below shows alternative paths merging and continuing along a single path. The diamond represents the point at which the paths converge.

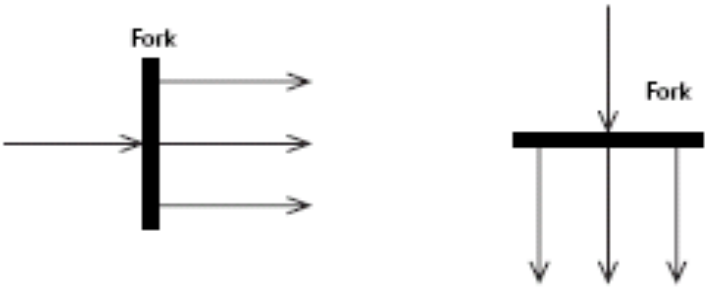


**Start and end**

The UML also provides icons to begin and end an Activity diagram. A solid dot indicates the beginning of the flow of activity. A bull's-eye indicates the end point. There may be more than one end point in an Activity diagram. Even the simplest Activity diagram typically has some decision logic that would result in alternative paths, each with its own unique outcome. If you really want to, you can draw all your arrows to the same end point, but there is no need to do so. Every end point means the same thing: Stop here. The start and end icons are shown below.

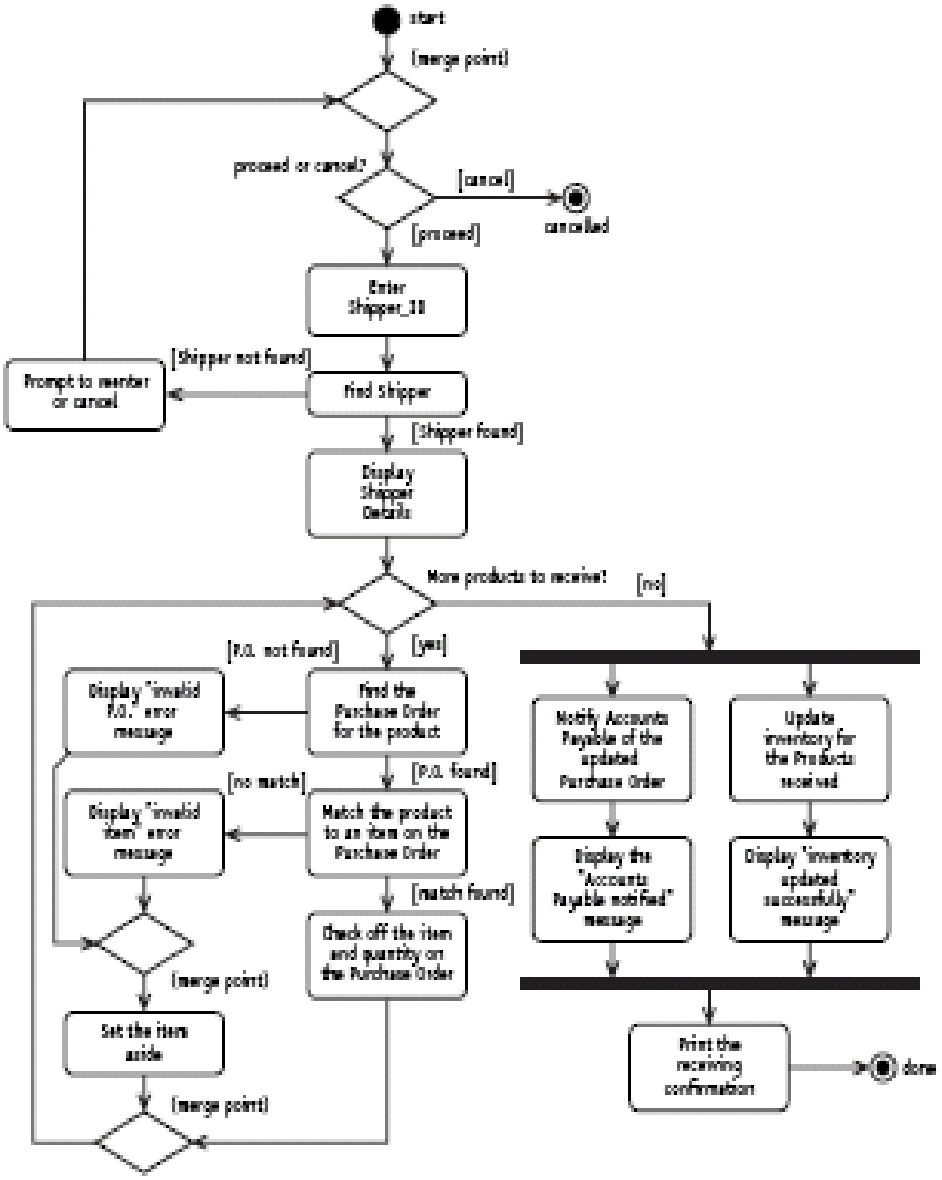

**Concurrency**

The UML notation for the Activity diagram also supports concurrency. This allows you to model the features of languages that have been introduced since the flowchart was invented, like Java, C++, and even Smalltalk, on hardware capable of true concurrency. To show that a single process starts multiple concurrent threads or processes, the UML uses a simple bar called a *fork.* In the examples in the figure below, you see one transition pointing at the bar and multiple transitions pointing away from the bar. Each outgoing transition is a new thread or process.




Synchronization or merging of the concurrent threads or processes is shown in much the same way. The figure below shows multiple transitions pointing at the bar, this time called a *synchronization bar,* and one pointing out of the bar. This indicates that the concurrent processing has ended and the process continues as a single thread or process.


The figure below shows the complete Activity diagram for the case study we earlier reviewed.


start
[merge point]
proceed or cancel?
[cancel]
cancelled
[proceed]
Enter Shipper_ID
[Shipper not found]
Prompt to reenter or cancel
Find Shipper
[Shipper found]
Display Shipper Details
More products to receive?
[no]
[yes]
[P.O. not found]
Display "invalid P.O." error message
Find the Purchase Order for the product
[P.O. found]
[no match]
Display "invalid item" error message
Match the product to an item on the Purchase Order
[match found]
Check off the item and quantity on the Purchase Order
[merge point]
Set the item aside
[merge point]
Notify Accounts Payable of the updated Purchase Order
Update inventory for the Products received
Display the "Accounts Payable notified" message
Display "inventory updated successfully" message
Print the receiving confirmation
done

# THE COPPERBELT UNIVERSITY

COMPUTER SCIENCE DEPARTMENT

**CS 432 – OBJECT ORIENTED SYSTEMS ANALYSIS AND DESIGN**

COMPILED BY DR NTALASHA

## Modeling the Dynamic View: The Sequence Diagram

The static view (Class and Object diagrams) represents how the objects are defined and arranged into a structure. It does not tell you how the objects behave when you put them to work. In contrast, the dynamic view represents the interactions of the objects in a system. The dynamic view contains diagrams specifically designed to model how the objects work together. It can represent how the system will respond to actions from the users, how it maintains internal integrity, how data is moved from storage to a user view, and how objects are created and manipulated.

**Understanding the Dynamic View**
Because system behaviors can be complex, the dynamic view tends to look at small, discrete pieces of the system like individual scenarios or operations. You may not see the dynamic view used as extensively as the Class diagram, simply because not all behaviors are complicated enough to warrant the extra work involved. Even so, the Class diagram and the diagrams of the dynamic view are the most often used diagrams in projects because they most directly reveal the specific features required in the final code.

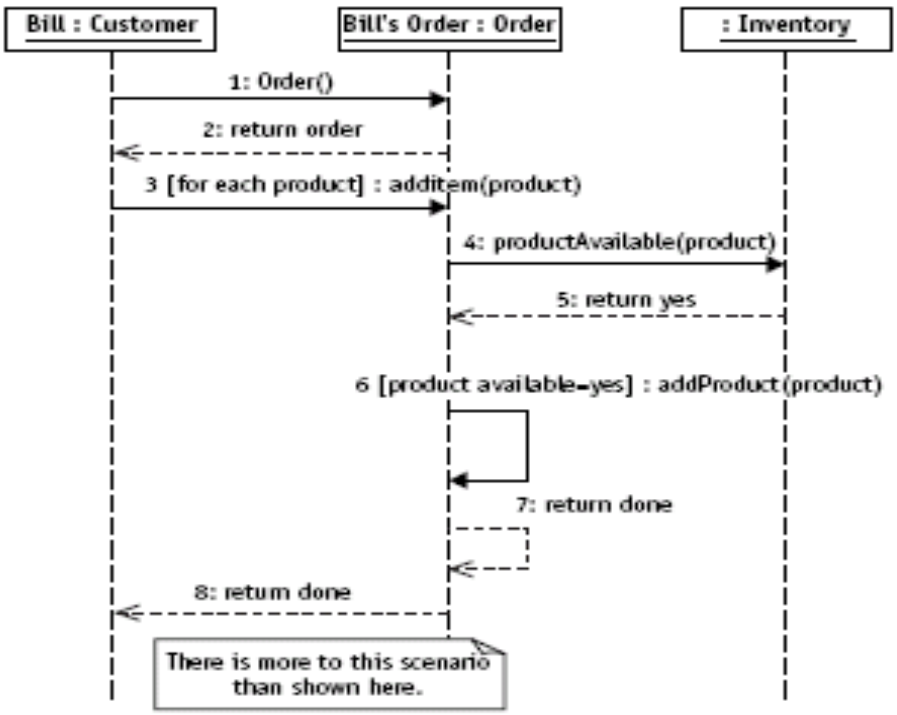
**Knowing the purpose of Sequence and Collaboration diagrams**
There are actually three UML diagrams in the dynamic view: the Sequence diagram, the Collaboration diagram and the Statechart diagram. The Sequence and Collaboration diagrams both illustrate the *interactions between objects.*Interactions show us how objects talk to each other. Each time that one object talks to another it talks to an interface (that is, it invokes an operation). So if you can model the interactions, you can find the interfaces/operations that the object requires.

In the Use Case view, you modeled the features of the system and developed scenarios describing how the system should behave when those features are used. The Sequence diagram provides a path from the textual descriptions of behaviors in the scenarios to operations/interfaces in the Class diagram.

**Mapping interactions to objects**
Everything in an object-oriented system is accomplished by objects. Objects take on the responsibility for things like managing data, moving data around in the system, responding to inquiries, and protecting the system. Objects work together by communicating or interacting with one another. The figure below shows a Sequence diagram with three participating objects: Bill the Customer, Bill's Order, and the Inventory. Without even knowing the notation formally, you can probably get a pretty good idea of what is going on.

- Steps 1 and 2: Bill creates an order.
- Step 3: Bill tries to add items to the order.
- Step 4 and 5: Each item is checked for availability in inventory.
- Step 6 and 7: If the product is available, it is added to the order.
- Step 8: He finds out that everything worked.

Building the Sequence diagram is easier if you have completed at least a first draft of the Use Case model and the Class diagram. From these two resources, you get sets of interactions (scenarios) and a pool of candidate objects to take responsibility for the interactions.



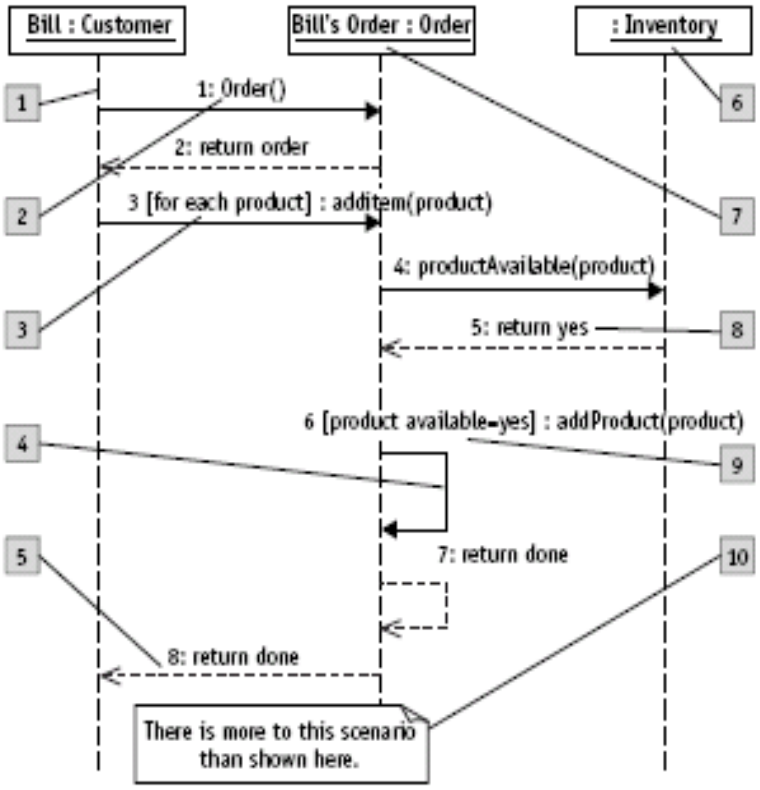

**Defining the basic notation of the Sequence diagram**

All Sequence diagrams are modeled at the object level rather than the class level to allow for scenarios that use more than one instance of the same class and to work at the level of facts, test data, and examples. The Sequence diagram uses three fundamental notation elements: objects, messages/stimuli, and object lifeline.

In the Sequence diagram, the *objects* use the same notation as in the Object diagram. In figure above, you see the three participating objects lined up across the top of the diagram. The object lifeline is a vertical dashed line below each object. The object lifeline always runs from the beginning at the top to the end at the bottom. The amount of time represented depends on the scenario or other behavior you're modeling.

A message or stimulus is usually a call, a signal, or a response. A message is represented by an arrow. The type of arrow visually describes the type of message. The solid line and solid arrowhead style represent a message that requires a response. The dashed arrows are the

responses. The messages are placed horizontally onto the timelines in relative vertical position to one another to represent the order in which they happen. This arrangement allows you to read the diagram from beginning to end by reading the messages from top to bottom.




The reference numbers on the figure above denote these items:

1. Object lifeline
2. Message/Stimulus
3. Iteration
4. Self-reference
5. Return
6. Anonymous object
7. Object name
8. Sequence number
9. Condition
10. Basic comment

The sequence numbers are optional but are very helpful when you need to discuss the diagram and make changes. Each message arrow describes an interface/operation on the object it is pointing to. Consequently, the message contains the operation signature, that is, the name, arguments, and optionally the return, such as addItem(product):boolean.

The dashed return arrows pointed to by references #2 and #5 each contain only the answer to a message. But the purpose of modeling is to reveal information, not make assumptions. Showing

the returns can help ensure that what you're getting back is consistent with what you asked for in the message.
Use the square condition brackets to enclose either the number of times or the condition that controls the repetitions, for example [for each product]. The same condition brackets may be used to control whether a message is even sent.

**Defining the extended notation for the Sequence diagram**
Sequence diagrams can be enhanced to illustrate object *activation* and object *termination* and to customize messages.
To show that an object is active, the notation is to widen the vertical object lifeline to a narrow rectangle. The narrow rectangle is called an "activation bar" or "focus of control."
Note that the object becomes active when it begins to do work.
To show that an object is terminated, place an X at the point in the object lifeline when the termination occurs. This is usually in response to a message such as delete or cancel.

# THE COPPERBELT UNIVERSITY

COMPUTER SCIENCE DEPARTMENT

**CS 432 – OBJECT ORIENTED SYSTEMS ANALYSIS AND DESIGN**

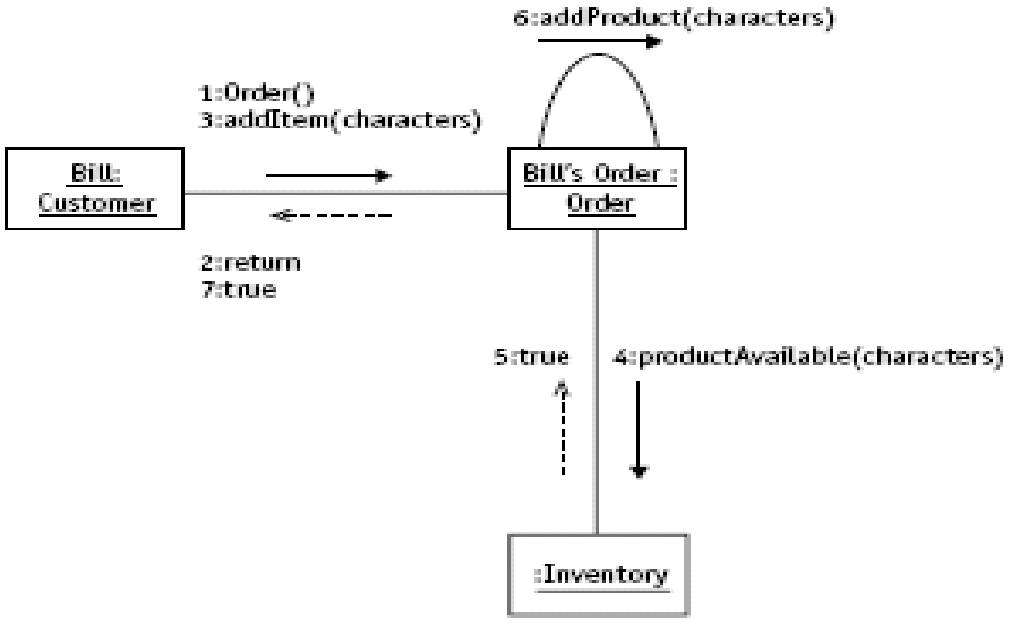
COMPILED BY DR NTALASHA

## Modeling the Dynamic View: The Collaboration Diagram

The Collaboration diagram offers an alternative to the Sequence diagram. Instead of modeling messages over time like the Sequence diagram, the Collaboration diagram models the messages on top of an Object diagram. The Collaboration diagram uses this approach in order to emphasize the effect of the object structures on the interactions.

### The Collaboration Diagram

The scenario (as shown in the figure below) shows the customer Bill creating an order and adding items to it, checking availability for each item as it is added. Just follow the numbered messages to step through the scenario. You can accomplish the same thing with both diagrams (that is, you can model the logical steps in a process like a Use Case scenario).




### Diagram similarities

Sequence and Collaboration diagrams model the same two elements: messages and objects.
In fact, the two diagrams are so similar that some modeling tools, like System Architect and Rational Rose, provide a toggle feature to switch back and forth between the two views. Like the Sequence diagram, the Collaboration diagram provides a tool for visually assigning responsibilities to objects for sending and receiving messages. By identifying an object as the receiver of a message, you are in effect assigning an interface to that object. It is kind of like receiving phone calls. You have to own the number the person is calling in order to receive the call. The number is your interface. The message description becomes an operation signature on the receiving object. The sending object invokes the operation.

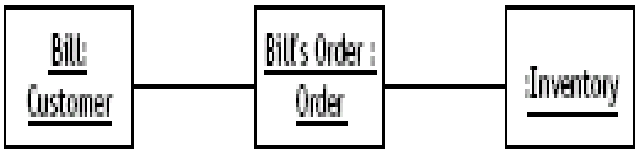
**Diagram differences**

The Collaboration diagram places a priority on mapping the object interactions to the object links (that is, drawing the participating objects in an Object diagram format and laying the messages parallel to the object links). This perspective helps validate the Class diagram by providing evidence of the need for each association as the means of passing messages. In contrast, the Sequence diagram does not illustrate the links at all. This highlights an advantage of the Collaboration diagram. Logically, you cannot place a message where there is no link because there is no physical avenue to send the message across. On a Sequence diagram there is nothing stopping you from drawing an arrow between two objects when there is no corresponding link. But doing so would model a logical interaction that cannot physically take place.

You can take the opposite view that drawing a message where there is no link reveals the *requirement* for a new link. Just make certain that you actually update your Class diagram or, you won't be able to implement the message illustrated on the diagram. An advantage of the Sequence diagram is its ability to show the creation and destruction of objects. Newly created objects can be placed on the timeline at the point where they are created. Sequence diagrams also have the advantage of showing object activation. Because the Collaboration diagram does not illustrate time, it is impossible to indicate explicitly when an object is active or inactive without interpreting the types of messages being passed.

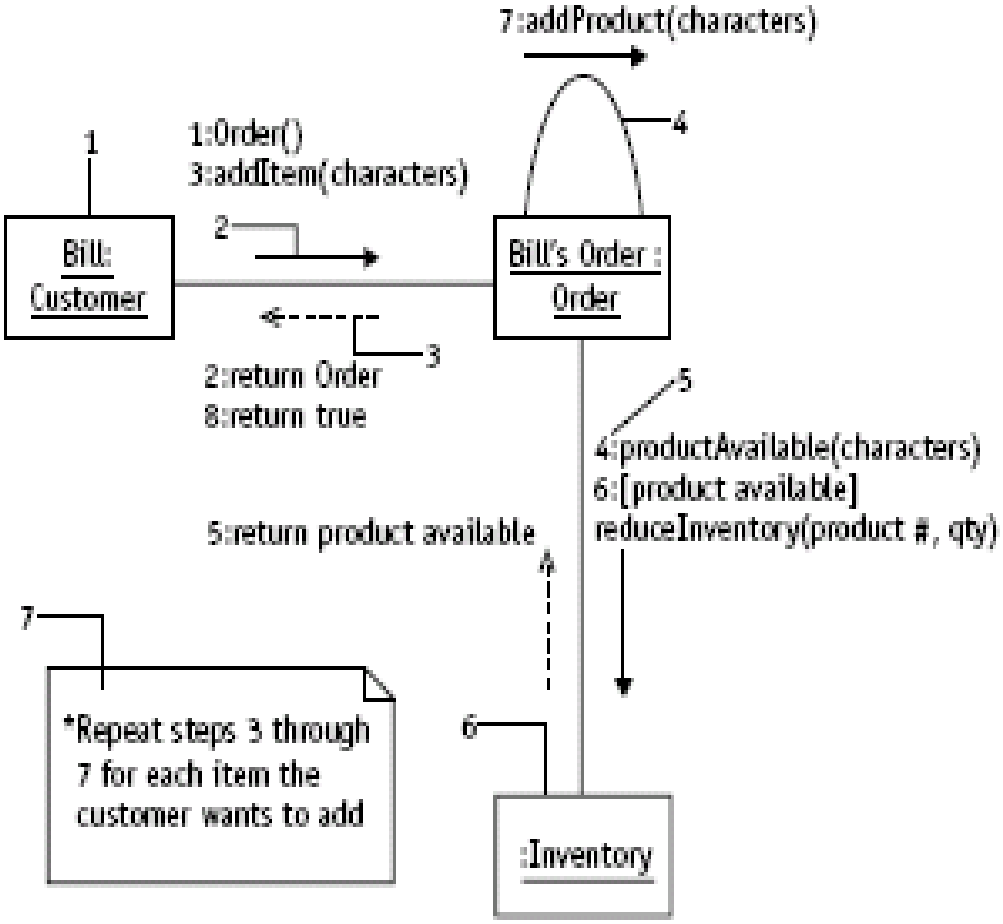
**Collaboration Diagram Notation**

The Collaboration diagram uses an Object diagram as its foundation. First, determine which objects will participate in the scenario. Draw the objects with only the name compartment, not the attributes. Then draw the links between them. Because any pair of classes can have more than one association, you need to use the Class diagram as your guide to identify the valid types of links that apply to the current sequence of messages. The figure below shows the objects and their links. You may leave the link names off of the links when there is only one type of association between the related classes. Add the names if there is more than one kind of link possible between the two objects and there is a need to clarify which relationship supports the interaction.



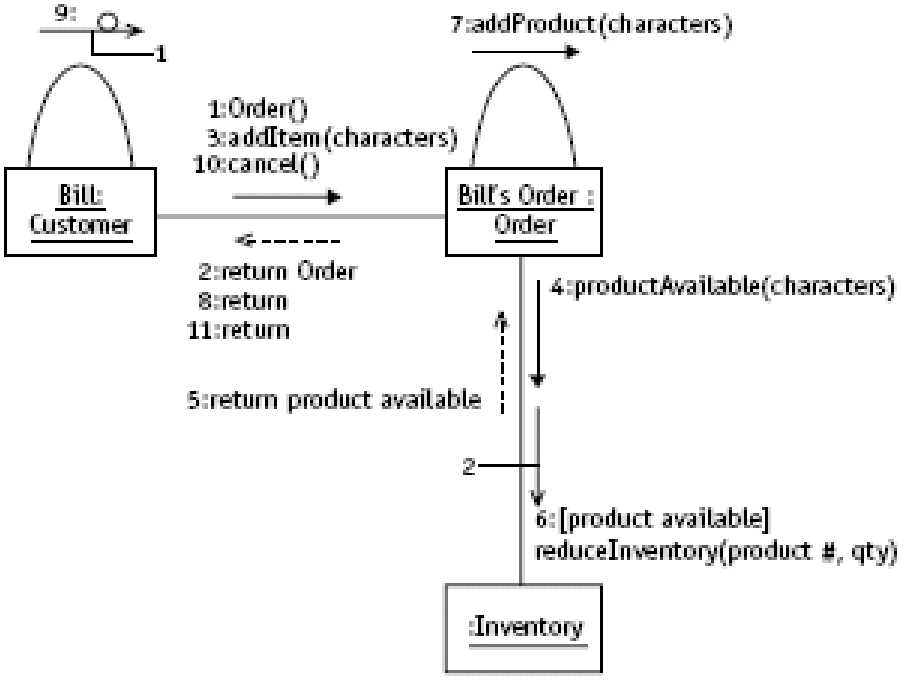

For each step of the scenario, draw the message arrow from the sending object to the receiving object. Place the message arrow parallel to the link between the sending and receiving objects. Having many messages placed on the same link is valid and, in fact, common as long as they really share the same message (arrow) type. Number the messages in the order in which they occur. The format for specifying a message is the same as on the Sequence diagram:
*Sequence-number Iteration : [condition] operation or return*

The figure below models the entire scenario for creating an order.




The following descriptions refer to the numbered items in the above figure so that you can see the notations used in context:

**1. Object:** This is a fully qualified object name, Bill, of the class Customer. The notation is exactly the same as on the Sequence diagram.
**2. Synchronous event or procedure call:** A synchronous event is a message that requires a reply, so you would expect to see a corresponding return message along the same link sometime later in the sequence. Procedure calls are simply another familiar way to describe this "ask and reply" form of interaction.
**3. Return:** Here is the return message for the message 1. Message 1 told the Order class to create a new Order object, Bill's Order. When the task of creating the object is completed, it passes back a reference to the requestor, Bill.
**4. Self-reference:** A self-reference is simply an object talking to itself saying something like, "It's time for me to get more coffee."
**5. Sequence number:** Because the Collaboration diagram has no way of showing the passage of time, it uses sequence numbers, like (4: ), to reveal the order of execution for the messages. There are no standards for the numbering scheme, so common sense and readability are your guides. The sequence numbers were optional on the Sequence diagram. They are required on the Collaboration diagram.
**6. Anonymous object:** Reference 6 shows another example of valid object notation. You do not have to name the instance if all you need to convey is that any object of this type (Inventory) would behave in this manner.
**7. Comment:** Reference 7 shows *one way* that you can reveal your intention to repeat a set of messages. As with Sequence diagrams, comments can be very helpful for explaining your intentions regarding iteration across a *set of messages* because the iteration notation provided by the UML works only for a single event.




Note the following items on the diagram:

**1. Timeout event:** I haven't labeled this message, mostly because a timeout would be a bit unusual for this type of scenario. This way you get to see the notation anyway. Actually, this is a common extension to the UML (that is, it isn't explicitly defined by the UML). The small circle represents a clock and sometimes even shows the clock hands within the circle. A timeout would be used for something like dialing into a network or polling a node on a network. If you don't get a response within a specified amount of time, you abandon the attempt and move on. In this case, if the Order doesn't respond within the specified time limit, the Order is cancelled in step 10.
**2. Asynchronous message:** An asynchronous message does not require a reply. Step 6 has been altered to simply tell the Inventory, "I've taken some of your stock. You might want to update your records. But I'm not going to wait around until you do."

COMPUTER SCIENCE DEPARTMENT

**CS 432 – OBJECT ORIENTED SYSTEMS ANALYSIS AND DESIGN**

COMPILED BY DR NTALASHA

# Modeling the Dynamic View: The Statechart Diagram

### Describing the Purpose and Function of the Statechart Diagram

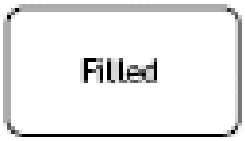
The Statechart describes the life of an object in terms of the events that trigger changes in the object's *state.* It identifies both the *external events* and *internal events* that can change the object's state. But what does that mean? The *state* of the object is simply its current condition. That condition is reflected in the values of the attributes that describe that object. There are behaviors in the system that alter those attribute values. A state describes an object, so it typically appears as an adjective in the problem description; for example, an account is open (an *open* account) or an account is overdrawn (an *overdrawn account).*

When the current condition, or state, of the account is overdrawn, the account will respond differently than when the account is in the open condition—checks will be rejected rather than paid or the bank will cover the check and charge you an exorbitant fee for its kindness.

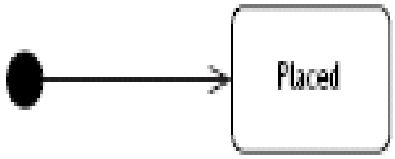
Next, contrast the scope of the Statechart with that of the Sequence diagram. The scope of the Statechart is the entire life of an object. The scope of the Sequence diagram is a single scenario. Consequently, it is possible to derive a Statechart from the set of Sequence diagrams that use the object. The Statechart models the events that trigger a *transition* (change) from one state to another state. Each event may have a corresponding *action* that makes the changes in the object (that is, alters the attribute values). While an object is in a state, it may also perform work associated with that state. Such work is called an *activity.*The Statechart can also be used to model concurrent activities within a state by creating parallel *substates* within a *superstate.* Using the substate and superstate notation, you can explicitly identify split and merge of control for concurrency.

### Defining the Fundamental Notation for a Statechart Diagram

The foundation for the Statechart is the relationship between states and events. The following examples illustrate the Statechart notation using the Order object. A state is modeled as a rounded rectangle with the state name inside, as shown in the figure below, much like the short form of the class icon, where only the name compartment is visible.



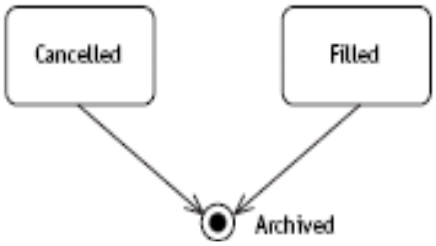
The initial state of an object has its own unique notation, a solid dot with an arrow pointing to the first state. The initial state indicates the state in which an object is created or constructed. You would read it as follows, "An Order begins in the 'Placed' state." In other words, the Order comes into existence when a customer places it.




Note that the initial state is the entire image in the figure above. It includes the dot, the arrow, and the state icon. In effect, the dot and arrow point to the first state. The Statechart event notation is a line style arrow connecting one state to another state. The arrow is actually the transition associated with the event. The direction of the arrow shows the direction of the change from one state to another. The figure below shows the event "products available" that causes the transition (the arrow) from the state "Placed" to the state "Filled."


An action is associated with an event. An action is the behavior that is triggered by the event and it is the behavior that actually changes the attributes that define the state of the object. To model the action, place a forward slash after the event name followed by the name of the action or actions you want performed, as in the figure below where the "products available" event triggers the fillOrder() action. The act of filling the Order alters its contents and redefines its state.
An action is an atomic task, and as such it cannot be broken into component tasks, nor can it be interrupted. There are no break points within it and, furthermore, stopping it midway would leave the object state undefined.


An object may reach a *final* state from which it may not return to an active state. In other words, you would never see an arrow going out of this state. A common usage is shown in the figure below. The Order may be archived from either state. But after it is archived, you may never change it. You may still see it and it may still exist, but you can no longer alter its state. The final state may also mean that the object has actually been deleted.



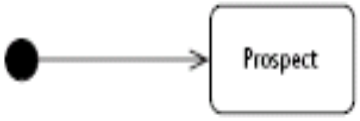

Because we tend to be cautious with our data, it is fairly rare that we literally delete data, so it is equally rare to see the final state. Often, even if an object is flagged for deletion or archive, you leave open the option to undo the deletion or archive to recover from an error or simply change your mind. In this situation, the deleted or archived state would be a normal state (the rounded rectangle).

**Building a Statechart Diagram**

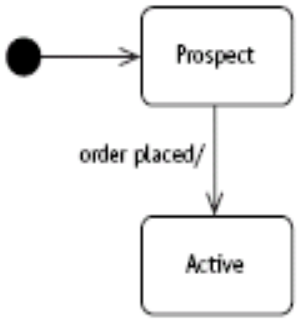
Now that you know the basic notation, you can step through the construction of a Statechart diagram. The problem statement describes your customers and how you view them for business purposes.

**Problem Statement**
We track current customer status to help avoid uncollectable receivables and identify customers worthy of preferred treatment. All customers are initially set up as prospects, but when they place their first order, they are considered to be active. If a customer doesn't pay an invoice on time, he is placed on probation. If he does pay on time and has ordered more than $10,000 in the previous six months, he warrants preferred status. Preferred status may be changed only if the customer is late on two or more payments. Then he returns to active status rather than probation, giving him the benefit of the doubt based on his preferred history.

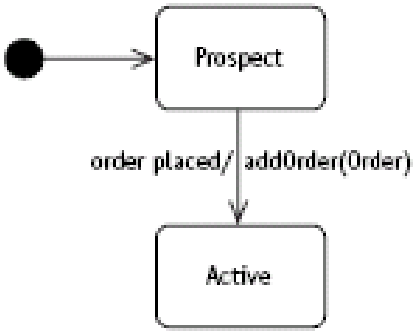
The first step is to identify the initial state of the customer. The problem statement told you "All customers are initially set up as prospects." To draw the initial state, you need three elements: the starting dot, the transition arrow, and the first state (Prospect). Thos is illustrated in the figure below with all three elements together.



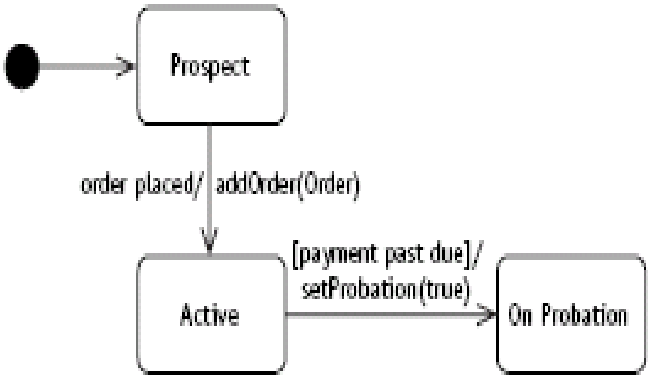

The next step is to identify an event that could change the prospect state to another state. The problem statement tells you, " . . . when they place their first order, they are considered to be active." To model the change you need at least the event that triggers the change, the transition arrow to show the direction of the change, and the new state that the object transitions to. The figure below shows all three elements added to the initial diagram.



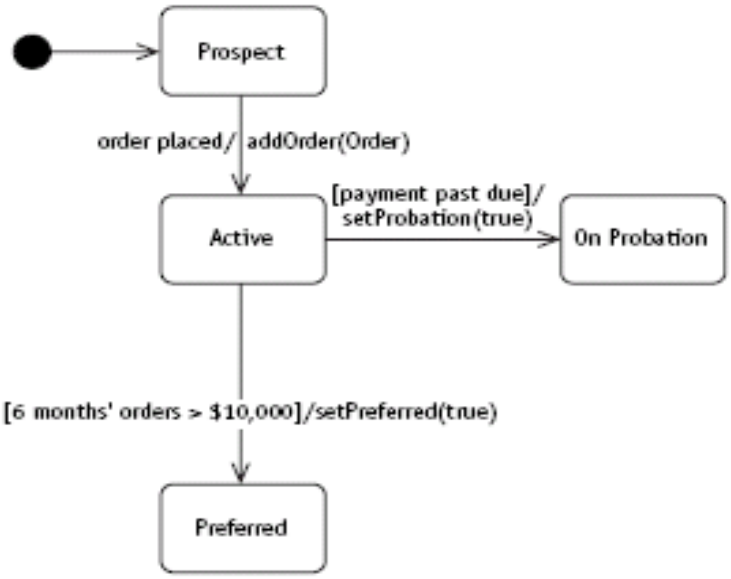

Now examine the event and determine what, if any, corresponding action needs to occur. The figure below shows the action addOrder(Order). This operation associates the newly placed order with the customer. According to the client's rules, this makes him an active customer.




The second event, payment past due, triggers the transition to On Probation. The figure below illustrates the new state, the event, the transition from active to on probation, and the action that actually makes the change to the object, setProbation(True).

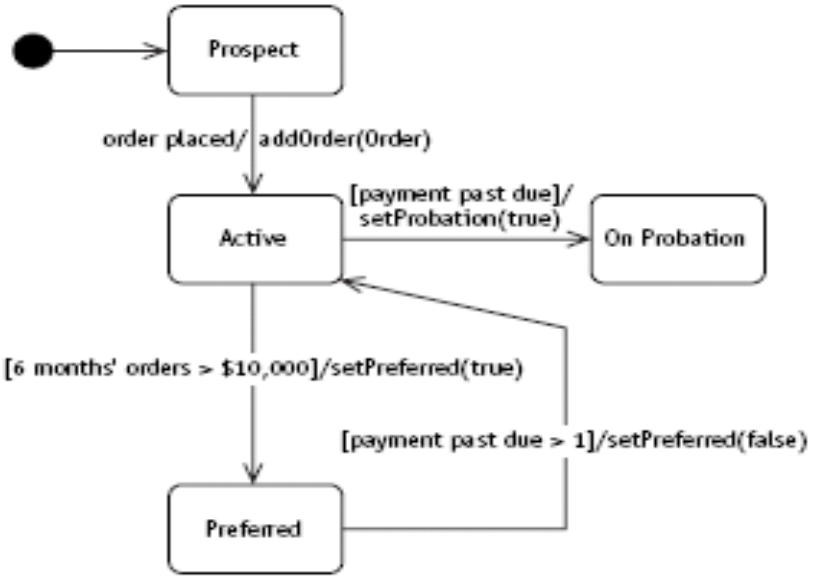



Watching the customer's performance generates the next event. "If he does pay on time and has ordered more than $10,000 in the previous six months, he warrants preferred status." The event is actually a condition that is met. The resulting action is to set the preferred status to true, setPreferred(True). The figure below adds the new state, the event, the transition, and the action.




Here is a good place to show how there may be more than one transition between the same two states. Although the example doesn't show it, you could add a second transition from *active* to *preferred* with the event, "The boss says give him preferred status so he will let the boss win at golf." You would have to draw a second transition and label it with the new event and the same action.

The last event addresses how a customer can fall out of preferred status. "Preferred status may be changed only if the customer is late on two or more payments." Again, the event is a condition and the response is an action that alters the state back to active. The figure below shows the transition back from Preferred to Active.

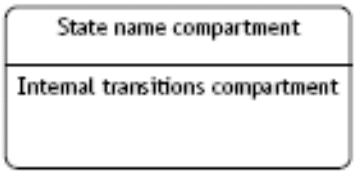



This Statechart did not have a final state because within the scope of the problem statement there is no time when a customer object can no longer change. "On Probation" might be a final state because there are no arrows coming out of it, but this happened only because of the limited size of the example. There is one more very important observation about events on a Statechart

diagram. The absence of an event is almost as informative as the presence of an event. In the figure above, the only events that cause a Customer to change from the active state are the conditions *6 months*' *orders > $10,000,* and *payment past due.* Even though you modeled the event *order placed* in another location, it has no effect when it happens to the Customer while he is in the active state. It simply is not recognized by the active state. You know this because there is no arrow leaving the active state in response to the *order placed* event. So the diagram reveals both the events that an object will respond to while in a state and the events it will not respond to.

**Defining Internal Events and Activities**

The state icon can also be expanded. The purpose of the expanded form is to reveal what the object can do while it is in a given state. The notation simply splits the state icon into two compartments: the *name compartment* and the *internal transitions compartment,* as illustrated in the figure below.




The internal transitions compartment contains information about actions and activities specific to that state. *Activities* are processes performed within a state. An activity tends not to be atomic, that is, an activity may be a group of tasks. Activities may be interrupted because they do not affect the state of the object. Contrast this with the earlier definition of an action, which said that you must not interrupt actions because they alter the state. Stopping an action midway could leave the object in an undefined state. Activities just do work. They do not change the state of the object. For example, the figure below models the active state of the Customer object. While in that state, the customer object generates a monthly invoice for the customer's purchasing activity and generates monthly promotions tailored to the Customer. To model activities within a state, use the keyword *Do:* followed by one or more activities.


These activities will be performed from the time the object enters the state until the object leaves the state or the activity finishes.

**THE COPPERBELT UNIVERSITY**

COMPUTER SCIENCE DEPARTMENT

**CS 432 – OBJECT ORIENTED SYSTEMS ANALYSIS AND DESIGN**

COMPILED BY DR NTALASHA

# Modeling the Static View: The Component Diagram

The Component diagram models the physical implementation of the software. The Deployment diagram models the physical architecture of the hardware (Deployment diagram). Combined, they model the integration and distribution of your application software across the hardware implementation.
Just as Class diagrams describe the organization and intent of your software design, components represent the physical implementations of your software design. The purpose of the Component diagram is to define software modules and their relationships to one another. Each component is a chunk of code that resides in memory on a piece of hardware. Each component must define an interface, which allows other components to communicate with that component. The interface and the internal implementation of the component are encapsulated in the classes that make up the component.
The UML groups components into three broad categories:

- Deployment components, which are required to run the system
- Work product components including models, source code, and data files used to create deployment components
- Execution components, which are components created while running the application

Components may depend on one another. For example, an executable (.exe) may require access to a dynamic link library (.dll), or a client application may depend on a server side application, which in turn depends on a database interface. Components may be dependent on classes. For example, to compile an executable file, you may need to supply the source classes. Given the key elements, component, component interface, and dependencies, you can describe the physical implementation of your system in terms of the software modules and the relationships among them.

### Defining the Notation for Components and Component Dependencies

A *component icon* is modeled as a rectangle with two small rectangles centered on the left edge. The name is placed inside the icon, as in the Figure 1.

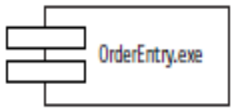



**Figure 1**

### Component stereotypes

*Component stereotypes* provide visual clues to the role that the component plays in the implementation. Some common component stereotypes include:

- <<**executable**>>: A component that runs on a processor
- <<**library**>>: A set of resources referenced by an executable during runtime
- <<**table**>>**:** A database component accessed by an executable
- <<**file**>>: Typically represents data or source code
- <<**document**>>: A document such as a page inserted into a Web page

These stereotypes refer to classifiers (implementations of the classes defined earlier in the process) and artifacts of the implementation of the classifiers, such as the source code, binary files, and databases.

**Component interfaces**

A component *interface* may be modeled in either of two ways. One way is to use a class with the stereotype <<**interface**>> attached to the component with a realization arrow, as shown in the figure 2 below. The realization arrow looks like the generalization symbol with a dashed line. To *realize* the interface means to apply it to something real like the executable.

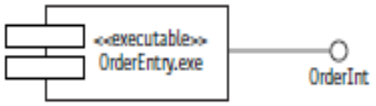



**Figure 2**

The interface implemented by a component is actually implemented by the classes within the component, so the interface should already have been defined in your Class diagrams. Also, a component may implement as many interfaces as it requires. The number and exact type of interfaces are dictated by the classes implemented by the component.

**Component Dependencies**

Dependencies between components are drawn with the dashed arrow from the dependent component to the component it needs help from. The same is true for component dependencies. In the Figure 3 below, the OrderEntry depends on the OrderEntry.exe component. The UML stereotype <<**becomes**>> means that the OrderEntry file literally becomes the OrderEntry executable at runtime. OrderEntry would be the code sitting on a storage device. At runtime it is loaded into memory and possibly even compiled. Then during execution the OrderEntry.exe component would depend on the three other components: orders.dll, inventory.dll, and orders.tbl.

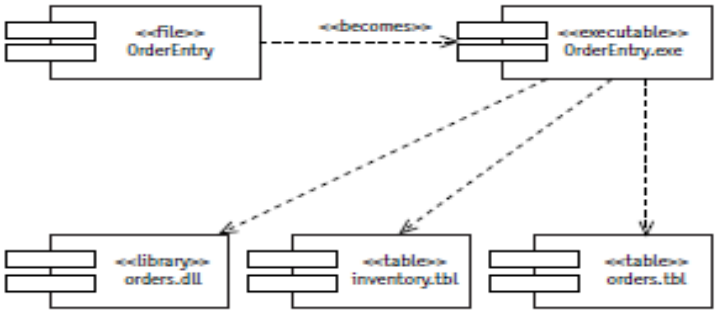



**Figure 3**

**Building a Component Diagram for the Case Study**

To review the notation for the Component diagram, I'll show you how to build one using each of the new model elements you just learned. The diagram will model the Receiving application. The application consists of two executable components, a shared interface, and three library components. The Receiving application consists of the classes that implement the Use Case ReceiveProduct, the server side application, and the client application (the UI). The other components represent the implementations of the classes used by receiving, Product, PurchaseOrder, and Inventory.

1. In Figure 4 below, create the Receiving.exe component. Name it and add the <<**executable**>> stereotype.

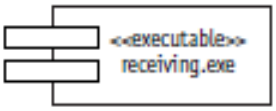



Figure 4

2. The Figure 5 below adds the purchaseorder.dll library component so that the Receiving component can validate incoming products against the purchase orders. The purchaseorder.dll component is the implementation of the PurchaseOrder class. It then draws a dependency from Receiving.exe to the purchaseorder.dll to show that the Receiving.exe needs help from the purchaseorder.dll to check the received products against the purchase order.

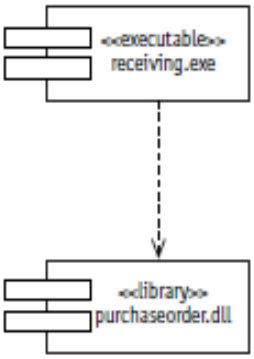



**Figure 5**

3. The Figure 6 below adds two more resource components. The product.dll component allows the Receiving application to update the product status to received. The inventory.dll component supports checks on the availability of locations where they can put the new product. It then adds the dependency from the Receiving.exe to the product.dll to show that the Receiving.exe needs access to the product.dll, and the dependency from the Receiving.exe to the inventory.dll to show that the Receiving.exe needs access to the inventory.dll in order to update inventory.

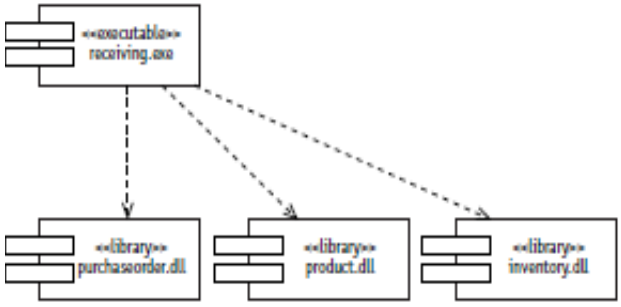



Figure 5

4. Figure 6 adds the client application that manages the user interface. The Receiving application provides the PO (or Purchase Order) interface. Figure 6 models the interface using the lollipop notation. The user interface application (ui.exe) accesses the Receiving application using the PO interface. This access is modeled as a dependency from ui.exe to the PO lollipop style interface to illustrate that the ui.exe will not work properly unless it can access the receiving application through the PO interface.

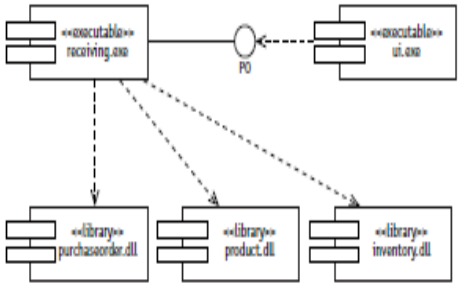



**Figure 6**

## Mapping the Logical Design to the Physical Implementation

Making components from classes involves choices about how to assemble these classes into cohesive units. The interfaces of the classes in the component make up the interface to the component. Figure 8 shows a database table component, orders.tbl, which implements the classes that define an order, namely Order, LineItem, and Product, and their association.

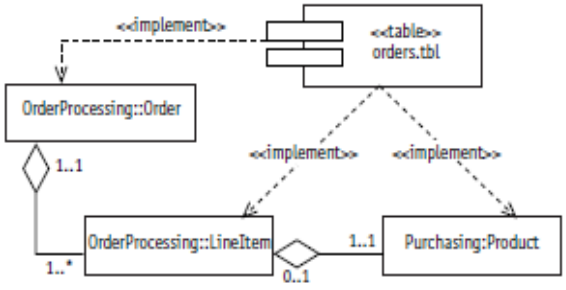



**Figure 8** *A component is created from classes*

In like manner, the main program in an application may implement some or all of the key classes in the logical model. To create the executable in Figure 9, you compile the classes together into a single executable.

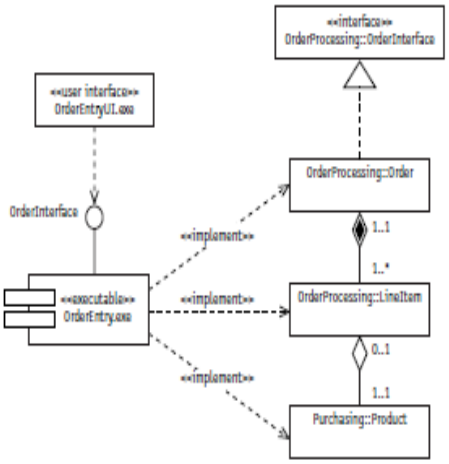



**Figure 9:** The OrderEntry.exe is created from multiple source classes

Quite often, however, a component consists of a single class implemented as an executable, file, library, table, or document. In Figure 10, the order entry executable references a set of library components for the individual classes rather than compiling the classes into one component. The user interface application is broken into two html components. The result is a more modular design.

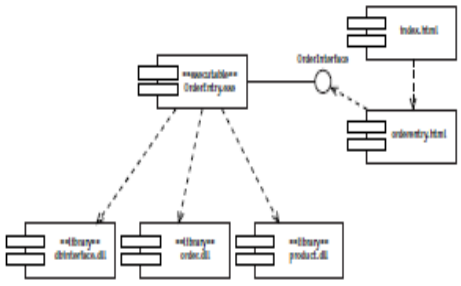



**Figure 10:** ***One class equals one component***

Finally, components can be organized into packages just like other diagrams and model elements. This can be very helpful when managing application distribution. The result is a directory containing all the software elements needed to implement the system or subsystem represented by the package

**THE COPPERBELT UNIVERSITY**

COMPUTER SCIENCE DEPARTMENT

**CS 432 – OBJECT ORIENTED SYSTEMS ANALYSIS AND DESIGN**

COMPILED BY DR NTALASHA

# Modeling the Static View: The Deployment Diagram

When the logical design is completed, the next step is to define the physical implementation of your design. The physical implementation must address three different problems: the software, the hardware, and the integration of the two. The Component diagram, is used to model the physical implementation of the software. The Deployment diagram is used to model the physical architecture of the hardware. Combined, they model the distribution of your application software across the hardware implementation.

**Describing the Purpose and Function of the Deployment Diagram**

The Deployment diagram describes the physical resources in much the way a Class diagram describes logical resources. The focus of the Deployment diagram is the nodes on which your software will run. Each node is a physical object that represents a processing resource. Most often this means a computer of some type, but it may mean a human resource for manual processes.

Each node contains, or is responsible for, one or more software components or objects. The software components on different nodes can communicate across the physical associations between the nodes.

The purpose of a Deployment diagram is to present a static view, or snapshot, of the implementation environment. A complete description of the system will likely contain a number of different Deployment diagrams, each focused on a different aspect of the system management. For example:

- One diagram might focus on how software components are distributed, such as where the source code resides and where it is shipped for implementation.
- Another diagram might model how the executable is loaded from one node to another node where it actually runs.
- For a multi-tiered application, the Deployment diagram would model the distribution of the application layers, their physical connections, and their logical paths of communication.

**Defining the Notation for the Deployment Diagram**

By now, the pattern for these physical diagrams should be getting pretty familiar (that is, resources and connections). Just like the Package and Component diagrams, the Deployment diagram has two types of elements: nodes (resources) and associations (connections).

The *node icon* is drawn as a 3D box (the shading is not necessary). Figure 1 models four types of nodes: Server, Client, Database Server, and Printer. The lines between the nodes are physical *associations* that are represented as a solid line from one node to another. Use multiplicity notation to define the number of nodes on each end of the association. For example, Figure 1

says that each Server is connected to one or more Client nodes, and each Client node is connected to exactly one Server node.

Naming the node associations poses an interesting problem. Because all the associations are physical connections, they could all end up with the same name, "connects to." Instead, you may want to use stereotypes to describe types of connections. Figure 1 says that the Server node and Client nodes are connected by an Ethernet connection using the **<<Ethernet>>** stereotype.

The node is a classifier (like classes, Use Cases, and components), so it can have attributes and specify behaviors in terms of the executables it deploys. Figure 2 shows an object-level view of a Deployment diagram. The object-level diagram models instances of each node just as an Object diagram models real entities. The name compartment on top identifies the node name and type, as well as the optional stereotype. The attribute compartment in the middle defines the properties of the node. The operations compartment at the bottom defines the components that run on the node.

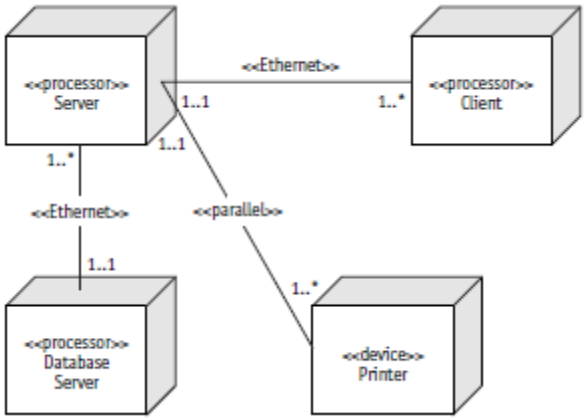



**Figure 1:** *Component diagram with four nodes and three associations*

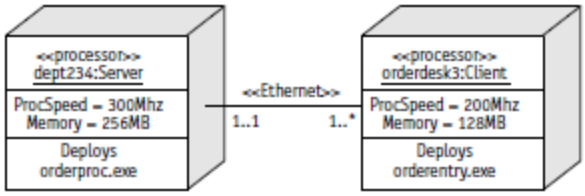



**Figure 2:** *An object-level Deployment diagram*

**THE COPPERBELT UNIVERSITY**

COMPUTER SCIENCE DEPARTMENT

**CS 432 – OBJECT ORIENTED SYSTEMS ANALYSIS AND DESIGN**

COMPILED BY DR NTALASHA

## Business Modeling

All businesses make some use of information technology, and it is important that their systems are really built to support the businesses of which they are an integrated part. The business is what ultimately defines the requirements on the information systems, and creating software without a proper understanding of the context in which that software is to operate is a dangerous adventure.

In order to get such an understanding, it is essential to make a model of the business. A *model* is a simplified view of a complex reality. It is a means to creating abstraction, allowing you to eliminate irrelevant details and focus on one or more important aspects at a time. Effective models also facilitate discussions among different stakeholders in the business, allowing them to agree on the key fundamentals and to work towards common goals. Finally, a business model can be the basis for other models, such as models for different information systems that support the business. Modeling (e.g., with UML) has been accepted and established as a means of analyzing and designing software. In order to create the best software, the businesses in which the software systems operate must also be modeled, understood, and sometimes improved. The business model is the center for conducting business or improving how the business is operated. The evolving models also help the developers structure and focus their thinking. Working with the models increases their understanding of the business and, hopefully, also their awareness of new opportunities for improving business.

Many development processes that use UML advocate that the system development should start with *use case modeling* to define the functional requirements on the system. A *use case* describes a specific usage of the system by one or more *actors.* An actor is a role that a user or another system has.

The objective of use case modeling is to identify and describe all the use cases that the actors require from the system. The use case descriptions then are used to analyze and design a robust system architecture that realizes the use cases (this is what is referred to as "use case driven" development). But how do you know that all of the use cases, or even the correct use cases that best support the business in which the system operates, are identified? To answer such questions you need to model and understand the system's surroundings.

Modeling a business's surroundings involves answering such questions as:

- How do the different actors interact?
- What activities are part of their work?
- What are the ultimate goals of their work?
- What other people, systems, or resources are involved that do not show up as actors to this specific system?
- What rules govern their activities and structures?
- Are there ways actors could perform more efficiently?

The answers to these questions come from tackling the entire business and looking beyond the functions of the information system currently being built (and using techniques other than use case modeling). The ultimate objective of all software systems is to give correct and extensive support to the business of which it is a part. However, when modeling the surroundings of the information system, you are no longer modeling software. Enter the world of *business modeling.*
There can be many reasons for doing business modeling:
**To better understand the key mechanisms of an existing business.**
The models can be used to train people by providing a clear picture of their role and tasks in the overall organization.
**To act as the basis for creating suitable information systems that support the business.**
The descriptions of the business are used to identify necessary information system support. The models are also used as a basis for specifying the key requirements on those systems. Ideally large parts of the business model can be mapped directly onto software objects. As more and more infrastructure software systems are bought, there is a potential for the systems that are developed to become more business driven where the developers can concentrate more on functionality that supports the business rather than solving technical incompatibilities or problems.

- **To act as the basis for improving the current business structure and operation.**
  The models identify changes in the current business that is necessary to implement the improved business model.
- **To show the structure of an innovated business.**
- **The model becomes the basis for the action plan.**
  Innovation suggests that radical change, rather than incremental changes, have been made to the business processes.
- **To experiment with a new business concept, or to copy or study a concept used by a competitive company (e.g., benchmarking on the model level).**
  The developed model becomes a sketch of a possible development for the business. The model can be a new idea, inspired by modeling other businesses, or taking advantage of new technologies, such as the Internet.
- **To identify outsourcing opportunities.**
  Parts of the business that are not considered the "core business" are delegated to outside suppliers. The models are used as the specification for the suppliers.

**Business Modeling with UML**

UML has quickly been adopted as the standard modeling language for modeling software systems.
UML was defined to model the architecture of software systems. Even though there are similarities between software and business systems, there are also some differences.
Business systems have many concepts that are never intended or suitable to execute in a program, such as the people working in the business, manufacturing production equipment, and rules and goals that drive the business processes. UML was initially designed to describe aspects of a software system. Because of this, UML needed to be extended in order to more clearly identify and visualize the important concepts of processes, goals, resources, and rules of a business system. To address this issue, we have created a set of extensions based on the existing model elements of UML. These extensions form a basic framework for business extensions to

UML (rather than a definitive set of business extensions) to which a business architect can add stereotypes or properties suitable to his or her line of business.
The standard extension mechanisms in UML that allow you to adapt UML to accommodate new concepts are:

- **Stereotypes.** An extension of the vocabulary of the UML, which allows you to create new building blocks from existing ones but specific to your problem [Booch, 1998]. Stereotypes may have their own visual icons that replace the icon which the existing UML element uses.
- **Tagged values (properties).** An extension of the properties of a UML element, which allows you to create new information in that element's specification [Booch,1998].
- **Constraints.** An extension of the semantics of a UML element, allowing you to add new rules or modify existing ones [Booch, 1998].
- Although different businesses have different goals and internal structures they use similar concepts to describe their structure and operation, and it is to represent these concepts the extension mechanisms in UML have been used for this purpose. The primary concepts used when defining the business system
- **Resources.** The objects within the business, such as people, material, information, and products, that are used or produced in the business. The resources are arranged in structures and have relationships with each other. Resources are manipulated (used, consumed, refined, or produced) through processes. Resources can be categorized into physical, abstract, and informational (each having their own stereotype).
- **Processes.** The activities performed within the business in which the state of business resources changes. Processes describe how the work is done within the business. Processes are governed by rules.
- **Goals.** The purpose of the business, or the outcome the business as a whole is trying to achieve. Goals can be broken down into sub-goals and allocated to individual parts of the business, such as processes or objects. Goals express the desired states of resources and are achieved by processes. Goals can be expressed as one or more rules.
- **Rules.** A statement that defines or constrains some aspect of the business and represents business knowledge. It governs how the business should be run (i.e., how the processes should execute) or how resources may be structured and related to each other. Rules can be enforced on the business from the outside by regulations or laws, or they can be defined within the business to achieve the goals of the business. Business rules are defined using the Object Constraint Language (OCL) which is a part of the UML standard.

Extensions use four different views of a business, and they are:

- **Business Vision View.** The overall vision of the business. This view describes a goal structure for the company, and illustrates problems that must be solved in order to reach those goals.
- **Business Process View.** The business processes that represent the activities and value created in the business. This view illustrates the interaction between the processes and resources in order to achieve the goal of each process, as well as the interaction between different processes.
- **Business Structural View.** The structures among the resources in the business, such as the organization of the business or the structure of the products created.

- **Business Behavioral View.** The individual behavior of each important resource and process in the business model and how they interact with each other. The views are not separate models; they are different perspectives on one or more specific aspect of the business. Combined, the views create a complete model of the business. Business Rules, defined in the OCL language, can be applied in all of the views.

**Business Vision View**

The *Business Vision View* depicts the company's goals. It is an image of where the company is headed. This view sets up the overall strategy for the business, defines the goals of the business, and acts as a guide for modeling the other views. The ultimate result of the Business Vision View is a definition of the desired future state of the company, and how that state can be reached. The primary result is expressed in a vision statement, one or more goal/problem models, and sometimes also a conceptual model. A *vision statement* is a short text document that outlines the vision of the company some years into the future. The *goal/problem model* is a UML object diagram that breaks down the major goals of the business into sub-goals, and indicates the problems that stand in the way of achieving those goals. The *conceptual model* is a UML class diagram that defines important concepts and relationships in the business to create a common set of terminology.

**Business Process View**

The Business Process View is at the center of business modeling. Processes show the activities to achieve an explicit goal and their relationships with the resources participating in the process. Resources include people, material, energy, information, and technology, and can be consumed, refined, created, or used (i.e., act as a catalyst) during the process. There are relationships between a process and its resources, between different processes that interact, and there is a coupling of processes to goals. A process diagram (based on a UML activity diagram) can also show how business events are generated or received between different processes, (i.e., as a means to interact or communicate between processes).

**Business Structural View**

The *Business Structural View* shows the structures of the resources, the products, or the services, and the information in the business, including the traditional organization of the company (divisions, departments, sections, business units, etc.). Traditional organizational charts and descriptions, and descriptions of the products and services the company provides are the basis for the Business Structural View. The information from the Process View is also used since it shows what resources are used. Note that typically these two views are modeled in parallel, since they contribute to each other and must be consistent. The view shows the structures of resources, of information and of the business organization.

**Business Behavioral View**

The *Business Behavioral View* illustrates both the individual behaviors of resources and processes in the business as well as the interaction between several different resources and processes. The behavior of the resource objects is governed by the Business Process View, which shows the overall main control flow of the work performed. However, the Business Behavioral View looks into each of the involved objects in more detail: their state, their behavior in each

state, and possible state transitions. The Behavioral View also shows the interaction between different processes, such as how they are synchronized with each other. By doing so, the Behavioral View is an important tool to use when allocating the exact responsibility for different activities, and when defining the exact behavior of each resource that takes part in each process.
The Behavioral view makes use of state chart diagrams, sequence diagrams and collaboration diagrams. Process diagrams can also be used to show interaction between processes.

**From Business Model to Software**

A common use of business models is to use them as the basis for identifying functional and non-functional requirements on one or more information systems. Business modeling is the way to actually know if the use cases defined for a system are the "right" or "optimal" requirements on that system. There are also other uses of the business model, because many of the objects and relationships found in the business model will also be objects and relationships in the information system model. It is important to realize that it is not a one-to-one mapping though, a critical analysis must be made of the business model to see what is applicable for a specific information system.
The ultimate goal is of course to create the software system(s) that best supports and fits into the business. The business model is used in software modeling to:

- **Identify the information systems that best support the operation of the business.** The systems can be new systems, standard systems, or legacy systems.
- **Find functional requirements.** The business model is used as a basis to identify the correct set of functions or use cases that the system should supply to the business processes.
- **Find non-functional requirements.** These requirements, such as robustness, security, availability, and performance, typically span and involve the entire system. They are often generic and not attached to a specific use case.
- **Act as a basis for analysis and design of the system.** For example, information about resources in the business model can be used to identify classes in the system. However, it is not possible to directly transfer the classes in the business model to the software model.
- **Identify suitable components.** Modern software development makes use of components: autonomous packages of functionality that are not specific to a certain system but can be used in several systems.

Most of component technology has concentrated on technical components, but there has been an increasing interest in defining business components that encapsulate a specific and reusable area of business functionality. Business models are a good way to identify areas of functionality and to define the appropriate set of services.

**THE COPPERBELT UNIVERSITY**

COMPUTER SCIENCE DEPARTMENT

**CS 432 – OBJECT ORIENTED SYSTEMS ANALYSIS AND DESIGN**

COMPILED BY DR NTALASHA

**Requirements Modeling**

**Requirements elicitation** focuses on describing the purpose of the system. The client, the developers, and the users identify a problem area and define a system that addresses the problem. Such a definition is called a **requirements specification** and serves as a contract between the client and the developers. The requirements specification is structured and formalized during analysis (Chapter 5, *Analysis*) to produce an **analysis model**. Both requirements specification and analysis model represent the same information. They differ only in the language and notation they use; the requirements specification is written in natural language, whereas the analysis model is usually expressed in a formal or semiformal notation. The requirements specification supports the communication with the client and users. The analysis model supports the communication among developers. They are both models of the system in the sense that they attempt to represent accurately the external aspects of the system. Given that both models represent the same aspects of the system, requirements elicitation and analysis occur concurrently and iteratively

Requirements elicitation and analysis focus only on the user's view of the system. For example, the system functionality, the interaction between the user and the system, the errors that the system can detect and handle, and the environmental conditions in which the system functions are part of the requirements. The system structure, the implementation technology selected to build the system, the system design, the development methodology, and other aspects not directly visible to the user are not part of the requirements.

3.1. Requirements elicitation includes the following activities:

- ***Identifying actors***. During this activity, developers identify the different types of users the future system will support.

- ***Identifying scenarios*.** During this activity, developers observe users and develop a set of detailed scenarios for typical functionality provided by the future system. Scenarios are concrete examples of the future system in use. Developers use these scenarios to communicate with the user and deepen their understanding of the application domain.
- ***Identifying use cases*.** Once developers and users agree on a set of scenarios, developers derive from the scenarios a set of use cases that completely represent the future system. Whereas scenarios are concrete examples illustrating a single case, use cases are abstractions describing all possible cases. When describing use cases, developers determine the scope of the system.
- ***Refining use cases***. During this activity, developers ensure that the requirements specification is complete by detailing each use case and describing the behavior of the system in the presence of errors and exceptional conditions.
- ***Identifying relationships among use cases*.** During this activity, developers identify dependencies among use cases. They also consolidate the use case model by factoring out common functionality. This ensures that the requirements specification is consistent.

- ***Identifying nonfunctional requirements*.** During this activity, developers, users, and clients agree on aspects that are visible to the user, but not directly related to functionality. These include constraints on the performance of the system, its documentation, the resources it consumes, its security, and its quality.

During requirements elicitation, developers access many different sources of information, including client-supplied documents about the application domain, manuals and technical documentation of legacy systems that the future system will replace, and most important, the users and clients themselves. Developers interact the most with users and clients during requirements elicitation. We focus on two methods

for eliciting information, making decisions with users and clients, and managing dependencies among requirements and other artifacts:

- **Joint Application Design (JAD)** focuses on building consensus among developers, users, and clients by jointly developing the requirements specification.
- **Traceability** focuses on recording, structuring, linking, grouping, and maintaining dependencies among requirements and between requirements and other work products.

## Requirements Elicitation Concepts

**In this section, we describe the main requirements elicitation concepts used in this chapter. In particular, we describe**

- Functional Requirements
- Nonfunctional Requirements

### Functional Requirements

**Functional requirements** describe the interactions between the system and its environment independent of its implementation. The environment includes the user and any other external system with which the system interacts.

### Nonfunctional Requirements

**Nonfunctional requirements** describe aspects of the system that are not directly related to the functional behavior of the system. Nonfunctional requirements include a broad variety of requirements that apply to many different aspects of the system, from usability to performance.

The FURPS+ model2 used by the Unified Process [Jacobson et al., 1999] provides the following categories of nonfunctional requirements:

- **Usability** is the ease with which a user can learn to operate, prepare inputs for, and interpret outputs of a system or component. Usability requirements include, for example, conventions adopted by the user interface, the scope of online help, and the level of user documentation. Often, clients address usability issues

by requiring the developer to follow user interface guidelines on color schemes, logos, and fonts.

- **Reliability** is the ability of a system or component to perform its required functions under stated conditions for a specified period of time. Reliability requirements include, for example, an acceptable mean time to failure and the ability to detect specified faults or to withstand specified security attacks. More recently, this category is often replaced by **dependability**, which is the property of a computer system such that reliance can justifiably be placed on the service it delivers. Dependability includes reliability, **robustness** (the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environment conditions), and **safety** (a measure of the absence of catastrophic consequences to the environment).
- **Performance** requirements are concerned with quantifiable attributes of the system, such as **response time** (how quickly the system reacts to a user input), **throughput** (how much work the system can accomplish within a specified amount of time), **availability** (the degree to which a system or component is operational and accessible when required for use), and **accuracy**.
- **Supportability** requirements are concerned with the ease of changes to the system after deployment, including for example, **adaptability** (the ability to change the system to deal with additional application domain concepts), **maintainability** (the ability to change the system to deal with new technology or to fix defects), and internationalization (the ability to change the system to deal with additional international conventions, such as languages, units, and number formats). The ISO 9126 standard on software quality [ISO Std. 9126], similar to the FURPS+ model, replaces this category with two categories: **maintainability** and **portability** (the ease with which a system or component can be transferred from one hardware or software environment to another). 2. FURPS+ is an acronym using the first letter of the

  requirements categories: Functionality, Usability, Reliability, Performance, and Supportability. The + indicates the additional subcategories. The FURPS model was originally proposed by [Grady, 1992].

**Fundamental requirements gathering techniques**

**Individually interview** people informed about the operation and issues of the current system and future systems needs

- **Interview groups** of people with diverse needs to fid synergies and contrasts among system requirements
- **Observe** workers at selected times to see how data are handled and what information people need to do their jobs
- **Study business documents**/**document analysis** to discover reported issues, policies, rules, and directions as well as concrete examples of the use of data and information in the org. What can the analysis of documents tell you about the requirements for a new system? In documents you can find information about the following:
    - Problems with existing systems (e.g., missing information or redundant steps
    - Opportunities to meet new needs if only certain information or information processing were available (e.g., analysis of sales based on customer type
    - Organizational direction that can influence information system requirements (e.g., trying to link customers and suppliers more closely to the organization)
    - Titles and names of key individuals who have an interest in relevant existing systems (e.g., the name of a sales manager who led a study of the buying behavior of key customers)
    - Values of the organization or individuals who can help determine priorities for different capabilities desired by different users (e.g., maintaining market share even if it means lower short-term profits)