

# Syllabus

## M.C.A.

### (Semester IV)

## Object Oriented Modeling and Design using UML

### INTRODUCTION

- An overview - Object basics - Object state and properties, Behavior, Methods, Messages.
- Object Oriented system development life cycle.
- Benefits of OO Methodology.

#### 1. Overview of Prominent OO Methodologies:

- a. The Rumbaugh OMT.
- b. The Booch methodology.
- c. Jacobson's OOSE methodologies.
- d. Unified Process.
- e. Introduction to UML.
- f. Important views & diagram to be modelled for system by UML.

#### 2. Factional view(models):

- **Use case diagram**
  - a. Requirement Capture with Use case.
  - b. Building blocks of Use Case diagram - actors, use case guidelines for use case models.
  - c. Relationships between use cases - extend, include, generalize.
- **Activity diagram**
  - a. Elements of Activity Diagram - Action state, Activity state, Object. node, Control and Object flow, Transition (Fork, Merge, Join)
  - b. Guidelines for Creating Activity Diagrams.
  - c. Activity Diagram - Action Decomposition (Rake ).
  - d. Partition - Swim Lane.

#### 3. Static structural view (Models):

- a. Classes, values and attributes, operations and methods, responsibilities for classes, abstract classes, access specification(visibility of attributes and operations).
- b. Relationships among classes: Associations, Dependencies., Inheritance - Generalizations, Aggregation.
- c. Adornments on Association: association names, association classes, qualified association, n-ary associations, ternary and reflexive association.
- d. Dependency relationships among classes, notations
- e. Notes in class diagram, Extension mechanisms, Metadata, Refinements, Derived , data, constraint, stereotypes, Package & interface notation.
- f. Object diagram notations and modeling, relations among objects (links).

#### 4. Class Modeling and Design Approaches:

- a. Three approaches for identifying classes - using Noun phrases, Abstraction, Use Case Diagram.

- b. Comparison of approaches.
- c. Using combination of approaches.
- d. Flexibility guidelines for class diagram: Cohesion, Coupling, Forms of coupling (identity, representational, subclass, inheritance), class Generalization, class specialization versus aggregation.

## **5. Behavioral (Dynamic structural view):**

- **State diagram**

- a. State Diagram Notations, events (signal events, change events, Time events).
- b. State Diagram states (composite states, parallel states, History states), transition and condition, state diagram behavior(activity effect, do-activity, entry and exit activity), completion transition, sending signals.

- **Interaction diagrams:**

- a. **Sequence diagram** - Sequence diagram notations and examples, iterations, conditional messaging, branching, object creation and destruction, time constraints, origin of links, Activations in sequence diagram.
- b. **Collaboration diagram** - Collaboration diagram notations and examples, iterations, conditional messaging, branching, object creation and destruction, time constraints, origin of links, activations in sequence diagram.

## **6. Approaches for developing dynamic systems:**

- a. Top - down approach for dynamic systems.
- b. Bottom - up approach for dynamic systems.
- c. Flexibility Guidelines for Behavioral Design - guidelines for allocating and designing behaviors that lead to more flexible design.

## **7. Architectural view:**

- a. Logical architecture: dependency, class visibility, sub systems.
- b. Hardware architecture: deployment diagram notations, nodes, object migration between node
- c. Process architecture: what are process and threads and their notations in UML, object synchronization, invocation schemes for threads ( UML notations for different types of invocations).
- d. Implementation architecture: component diagram notations and examples.

## **8. Reuse: Libraries, Frameworks components and Patterns:**

- a. Reuse of classes.
- b. Reuse of components.
- c. Reuse of frameworks, black box framework, white box frame.
- d. Reuse of patterns: Architectural pattern and Design pattern.

**Term Work / Assignment:** Each candidate will submit an approximately 10-page written report on a case study or mini project. Students have to do OO analysis & design for the project problem, and develop use case model, analysis model and design model for it, using UML.

**Reference books:**

1. Designing Flexible Object Oriented systems with UML - Charles Ritcher
2. Object Oriented Analysis & Design, Sat/.inger. Jackson, Burd Thomson
3. Object oriented Modeling and Design with UML - James Rumbaugh. Micheal Blaha (second edition)
4. The Unified Modeling Language User Guide - Grady Booch, James Rumbaugh, Ivar Jacobson.
5. Object Oriented Modeling and Design - James Rumbaugh
6. Teach Yourself UML in 24 Hours - Joseph Schmuilers
7. Object-Oriented Analysis and Design: using UML Mike O'Docherty Wiley Publication

**Practical assignment:** Nine assisjnments, one on each of the diagrams learnt in UML



# INTRODUCTION TO OBJECTS

## Unit Structure

- 1.1 Overview
- 1.2 Object state & behaviour
- 1.3 The Property (object attribute)
- 1.4 Object oriented system development life cycle
- 1.5 Advantages of Object Oriented Methodology

---

## 1.1 OVERVIEW

---

**Objects** are composite data types. An *object* provides for the storage of multiple data values in a single unit. Each value is assigned a name which may then be used to reference it. Each element in an object is referred to as a *property*. Object properties can be seen as an unordered list of name value pairs contained within the container object.

Object comes in two flavors. There are system defined objects, which are predefined and come with the JavaScript parser or with the browser running the parser. And there are user defined objects, which the programmer creates.

**class**: a definition, or description, of how the object is supposed to be created, what it contains, and how it work

There are two important concepts to understand when talking about objects. These are the ideas of class and instance.

Creating objects is a two step process. First you must define a class of objects, then you use the object class by declaring instances of that class within the program. The object *class* is a definition, or description, of how the object is supposed to be created, what it contains, and how it works. The object *instance* is a composite data type, or object, created based on the rules set forth in the class definition.

**instance**: a composite data type, or object, created based on the rules set forth in the class definition

This break between class and instance is not new, it is just that before objects, all data classes were hard coded into the parser and you could just make use of them while creating variables that were instances of those classes. Someone, somewhere, had to write the code to define the integer data type as being a numeric value with no fractional component. Whenever you declare an integer variable, you make use of this definition to create, or instantiate, an integer. Fortunately for us, it all happens behind the scenes.

The point of object-based programming languages is that they give the user the ability to define their own data types that can be specifically tailored to the needs of the application. There are still system-defined data types and object classes, so you don't need to

worry about defining commonly used types of variables, but you now can go beyond them.

Since objects are composite data types, they can contain more than one piece of data. In fact, the very point of the object is to bring together related data elements into a logical grouping. This grouping can contain not only data values, but also rules for processing those values. In an object, a data element is called a *property*, while the rules the object contains for processing those values are called *methods*. This makes objects very powerful because they can not only store data, but they can store the instructions on what to do with that data.

```
public class Student
{
}
```

According to the sample given below we can say that the *student* object, named *objectStudent*, has created out of the *Student* class.

```
Student objectStudent = new Student();
```

---

## 1.2 OBJECT STATE & BEHAVIOUR

---

Real-world objects share two characteristics: They all have *state* and *behavior*. Dogs have state (name, color, breed, hungry) and behavior (barking, fetching, wagging tail). Bicycles also have state (current gear, current pedal cadence, current speed) and behavior (changing gear, changing pedal cadence, applying brakes)

**State** : Every object, at any given point of time would have to have a set of attributes defining its State.

**Behavior** : Every object based on its state and optionally identity will have particular behavior.

---

## 1.3 THE PROPERTY (OBJECT ATTRIBUTE)

---

Properties are variables contained in the class; every instance of the object has those properties. Properties should be set in the prototype property of the class (function) so that inheritance works correctly.

Working with properties from within the class is done by the keyword *this*, which refers to the current object. Accessing (reading or writing) a property outside of the class is done with the syntax: *InstanceName.Property*; this is the same syntax used by C++, Java, and a number of other languages. (Inside the class the syntax *this.Property* is used to get or set the property's value.)

### The methods

Methods follow the same logic as properties; the difference is that they are functions and they are defined as functions. Calling a method is similar to accessing a property, but you add *()* at the end of the method name, possibly with arguments.

## The messages

An object-oriented program is a growing and shrinking collection of objects that interact via messages. You can send the same message to similar objects--the target decides how to implement or respond to a message at run-time. Objects can send and receive messages. Over its lifetime, the object has a state that varies.

A message is called a **request** or an **event**. The **event** contains the name of the object, the name of the operation, and maybe a group of parameters. As a result of receiving an event (message) the object runs a script (program) that may modify its state or send messages to other objects.

---

## 1.4 OBJECT ORIENTED SYSTEM DEVELOPMENT LIFE CYCLE

---

We live in a world of objects. These objects exist in nature, in man-made entities, in business, and in the products that we use. They can be categorized, described, organized, combined, manipulated and created. Therefore, an object-oriented view has come into picture for creation of computer software. An object-oriented approach to the development of software was proposed in late 1960s.

Object-Oriented development requires that object-oriented techniques be used during the analysis, and implementation of the system. This methodology asks the analyst to determine what the objects of the system are, how they behave over time or in response to events, and what responsibilities and relationships an object has to other objects. Object-oriented analysis has the analyst look at all the objects in a system, their commonalties, difference, and how the system needs to manipulate the objects.

### Object Oriented Process

The Object Oriented Methodology of Building Systems takes the objects as the basis. For this, first the system to be developed is observed and analyzed and the requirements are defined as in any other method of system development. Once this is done, the objects in the required system are identified. For example in case of a Banking System, a customer is an object, a chequebook is an object, and even an account is an object.

In simple terms, Object Modeling is based on identifying the objects in a system and their interrelationships. Once this is done, the coding of the system is done. Object Modeling is somewhat similar to the traditional approach of system designing, in that it also follows a sequential process of system designing but with a different approach. The basic steps of system designing using Object Modeling may be listed as:

- System Analysis
- System Design
- Object Design
- Implementation

## **System Analysis**

As in any other system development model, system analysis is the first phase of development in case of Object Modeling too. In this phase, the developer interacts with the user of the system to find out the user requirements and analyses the system to understand the functioning.

Based on this system study, the analyst prepares a model of the desired system. This model is purely based on what the system is required to do. At this stage the implementation details are not taken care of. Only the model of the system is prepared based on the idea that the system is made up of a set of interacting objects. The important elements of the system are emphasized.

## **System Design**

System Design is the next development stage where the overall architecture of the desired system is decided. The system is organized as a set of sub systems interacting with each other. While designing the system as a set of interacting subsystems, the analyst takes care of specifications as observed in system analysis as well as what is required out of the new system by the end user.

As the basic philosophy of Object-Oriented method of system analysis is to perceive the system as a set of interacting objects, a bigger system may also be seen as a set of interacting smaller subsystems that in turn are composed of a set of interacting objects. While designing the system, the stress lies on the objects comprising the system and not on the processes being carried out in the system as in the case of traditional Waterfall Model where the processes form the important part of the system.

## **Object Design**

In this phase, the details of the system analysis and system design are implemented. The Objects identified in the system design phase are designed. Here the implementation of these objects is decided as the data structures get defined and also the interrelationships between the objects are defined.

Let us here deviate slightly from the design process and understand first a few important terms used in the Object-Oriented Modeling.

As already discussed, Object Oriented Philosophy is very much similar to real world and hence is gaining popularity as the systems here are seen as a set of interacting objects as in the real world. To implement this concept, the process-based structural programming is not used; instead objects are created using data structures. Just as every programming language provides various data types and various variables of that type can be created, similarly, in case of objects certain data types are predefined.

For example, we can define a data type called pen and then create and use several objects of this data type. This concept is known as creating a class.

### **Class:**

A class is a collection of similar objects. It is a template where certain basic characteristics of a set of objects are defined. The class defines the basic attributes and the operations of the objects of that type. Defining a class does not define any object, but it only creates a template. For objects to be actually created instances of the class are created as per the requirement of the case.

### **Abstraction:**

Classes are built on the basis of abstraction, where a set of similar objects are observed and their common characteristics are listed. Of all these, the characteristics of concern to the system under observation are picked up and the class definition is made. The attributes of no concern to the system are left out. This is known as abstraction.

The abstraction of an object varies according to its application. For instance, while defining a pen class for a stationery shop, the attributes of concern might be the pen color, ink color, pen type etc., whereas a pen class for a manufacturing firm would be containing the other dimensions of the pen like its diameter, its shape and size etc.

### **Inheritance:**

Inheritance is another important concept in this regard. This concept is used to apply the idea of reusability of the objects. A new type of class can be defined using a similar existing class with a few new features. For instance, a class vehicle can be defined with the basic functionality of any vehicle and a new class called car can be derived out of it with a few modifications. This would save the developers time and effort as the classes already existing are reused without much change.

Coming back to our development process, in the Object Designing phase of the Development process, the designer decides onto the classes in the system based on these concepts. The designer also decides on whether the classes need to be created from scratch or any existing classes can be used as it is or new classes can be inherited from them.

### **Implementation**

During this phase, the class objects and the interrelationships of these classes are translated and actually coded using the programming language decided upon. The databases are made and the complete system is given a functional shape.

The complete OO methodology revolves around the objects identified in the system. When observed closely, every object exhibits some characteristics and behavior. The objects recognize and respond to certain events. For example, considering a Window on the screen as an object, the size of the window gets changed when resize button of the window is clicked. Here the clicking of the button is an event to which the window responds by changing its state from the old size to the new size.



While developing systems based on this approach, the analyst makes use of certain models to analyze and depict these objects. The methodology supports and uses three basic Models:

- **Object Model** - This model describes the objects in a system and their interrelationships. This model observes all the objects as static and does not pay any attention to their dynamic nature.
- **Dynamic Model** - This model depicts the dynamic aspects of the system. It portrays the changes occurring in the states of various objects with the events that might occur in the system.
- **Functional Model** - This model basically describes the data transformations of the system. This describes the flow of data and the changes that occur to the data throughout the system.

While the Object Model is most important of all as it describes the basic element of the system, the objects, all the three models together describe the complete functional system.

As compared to the conventional system development techniques, OO modeling provides many benefits. Among other benefits, there are all the benefits of using the Object Orientation. Some of these are:

- **Reusability** - The classes once defined can easily be used by other applications. This is achieved by defining classes and putting them into a library of classes where all the classes are maintained for future use. Whenever a new class is needed the programmer looks into the library of classes and if it is available, it can be picked up directly from there.
- **Inheritance** - The concept of inheritance helps the programmer use the existing code in another way, where making small additions to the existing classes can quickly create new classes.
- **Programmer has to spend less time and effort and can concentrate on other aspects of the system due to the reusability feature of the methodology.**
- **Data Hiding** - Encapsulation is a technique that allows the programmer to hide the internal functioning of the objects from the users of the objects. Encapsulation separates the internal functioning of the object from the external functioning thus providing the user flexibility to change the external behaviour of the object making the programmer code safe against the changes made by the user.
- **The systems designed using this approach are closer to the real world as the real world functioning of the system is directly mapped into the system designed using this approach**

---

## 1.5 ADVANTAGES OF OBJECT ORIENTED METHODOLOGY

---

- **Object Oriented Methodology closely represents the problem domain. Because of this, it is easier to produce and understand designs.**
- **The objects in the system are immune to requirement changes. Therefore, allows changes more easily.**

- Object Oriented Methodology designs encourage more re-use. New applications can use the existing modules, thereby reduces the development cost and cycle time.
- Object Oriented Methodology approach is more natural. It provides nice structures for thinking and abstracting and leads to modular design.



# 2

## OBJECT MODELING TECHNIQUE (OMT)

### Unit Structure

- 2.1 Introduction
- 2.2 The Rumbaugh OMT
- 2.3 The Booch OMT
- 2.4 Jacobson OOSE
- 2.5 Object-oriented software engineering life cycle
- 2.6 UNIFIED process model
- 2.7 Views in UML
- 2.8 UML Diagrams

---

### 2.1 INTRODUCTION

---

- The **object-modeling technique (OMT)** is an object modeling language for software modeling and designing.
- It was developed by Rumbaugh, Blaha, Premerlani, Eddy and Lorensen as a method to develop object-oriented systems, and to support object-oriented programming.

---

### 2.2 THE RAMBAUGH OMT

---

The purposes of modeling according to Raumbaugh (1991) are:

- ✧ testing physical entities before building them (simulation),
- ✧ communication with customers,
- ✧ visualization (alternative presentation of information), and
- ✧ reduction of complexity.

**The Rumbaugh OMT has proposed three main types of models:**

1. **Object model**: Main concepts are classes and associations, with attributes and operations. Aggregation and generalization are predefined relationships.
2. **Dynamic model**: The dynamic model represents a state/transition view on the model. Main concepts are states, transitions between states,

and events to trigger transitions. Actions can be modeled as occurring within states.

**3. Functional model :** The functional model handles the process of the model, corresponding roughly to data flow diagrams. Main concepts are process, data store, data flow, and actors.

*OMT is a predecessor of the Unified Modeling Language (UML).*

---

## 2.3 THE BOOCH OMT

---

- The analysis phase is split into steps.
- **Customer's Requirements Step:** The first step is to gather the requirements from the customer perspective. This analysis step generates a high-level description of the system's function and structure.
- **Domain analysis:** The domain analysis is done by defining object classes; their attributes, inheritance, and methods. State diagrams for the objects are then established.
- The analysis phase is completed with a **validation step**.
- The analysis phase iterates between the customer's requirements step, the domain analysis step, and the validation step until consistency is reached.
- Once the analysis phase is completed, the Booch methodology develops the architecture in the design phase.
- The design phase is **iterative**.
- A logic design is mapped to a physical design like processes, performance, data types, data structures, visibility are defined.
- A prototype is created and **tested**. The process iterates between the logical design, physical design, prototypes, and testing.
- The Booch software engineering methodology is sequential in the sense that the analysis phase is completed and then the design phase is completed.
- The methodology is cyclical in the sense that each phase is composed of smaller cyclical steps.
- **Drawbacks:**
- There is no explicit priority setting nor a non-monotonic control mechanism.
- The Booch methodology concentrates on the analysis and design phase and does not consider the implementation or the testing phase in much detail.

---

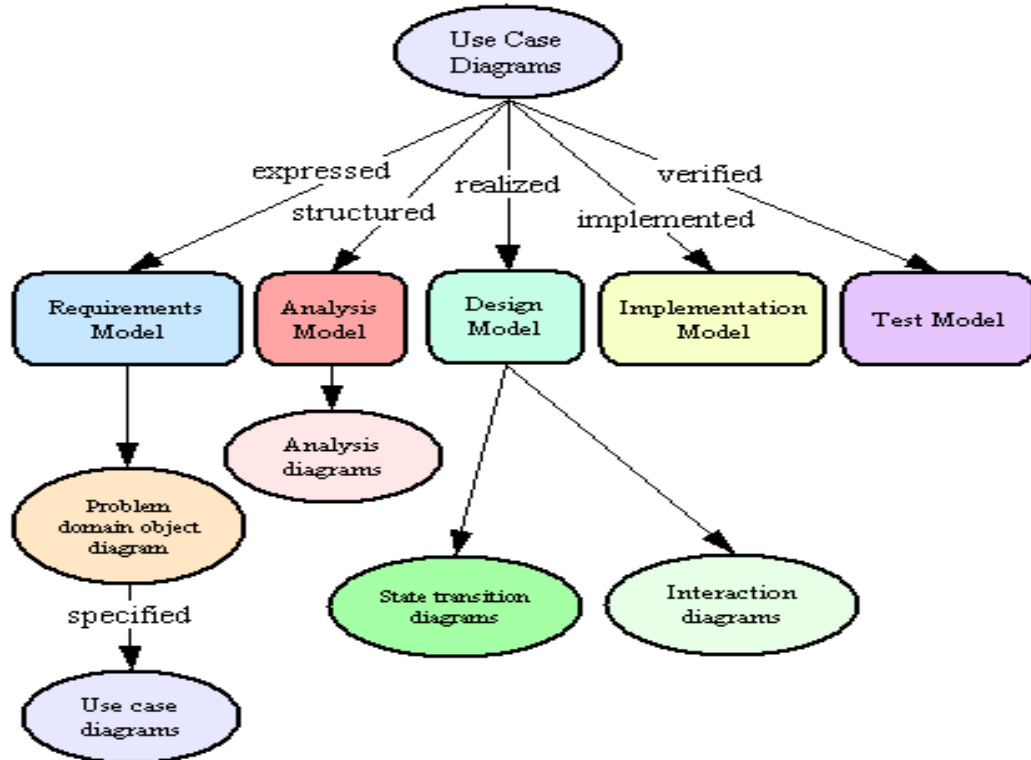
## 2.4 JACOBSON OOSE

---

- Object-Oriented Software Engineering (OOSE) is a software design technique that is used in software design in object-oriented programming.
- OOSE is developed by **Ivar Jacobson** in 1992.

- OOSE is the first object-oriented design methodology that employs use cases in software design.
- It includes requirements, an analysis, a design, an implementation and a testing model.

### Object oriented software engineering




---

## 2.5 OBJECT-ORIENTED SOFTWARE ENGINEERING LIFE CYCLE

---

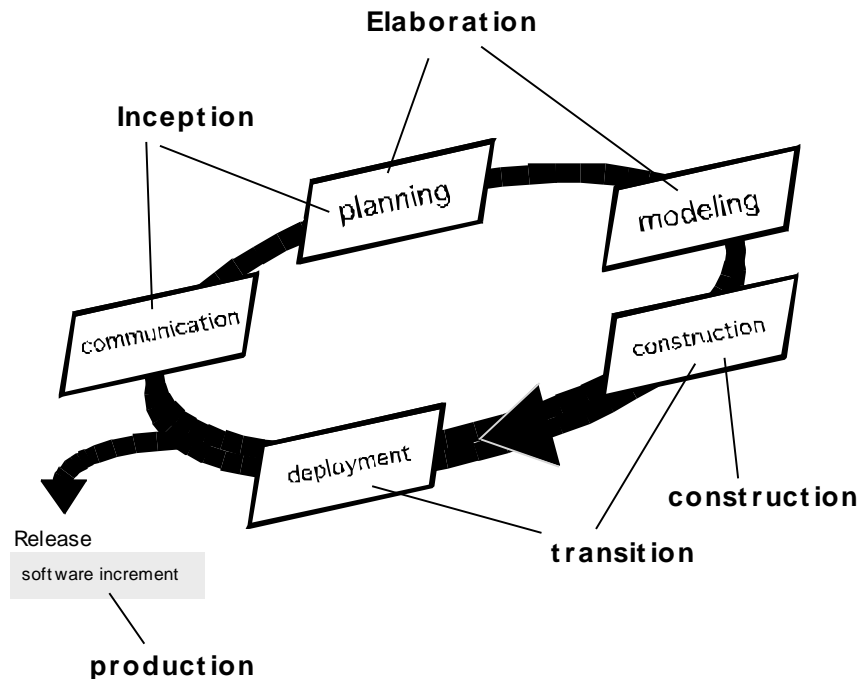
- **Requirements Engineering:**
  - requirements gathering,
  - Object oriented analysis and
  - specification
- **OO Design:**
  - architectural design
  - detailed design
  - both require the transformation of functional requirements into OO design elements
- **Implementation or Coding:**
  - Programming using OO programming languages and tools.
- **Testing:**
  - unit testing; test methods within each object

- integration testing; test collaborations between objects
- system testing; test the entire system as a collection of objects
- acceptance testing; test for standards and customer satisfaction

---

## 2.6 UNIFIED PROCESS MODEL

---



- **Iterative and Incremental:**
  - The Unified Process is an iterative and incremental development process.
  - The Elaboration, Construction and Transition phases are divided into a series of timeboxed iterations.
  - Each iteration results in an *increment*, which results in improved functionality .
- **Use Case Driven:**
  - In the Unified Process, use cases are used to capture the functional requirements and to define the contents of the iterations.
- **Risk Focused**
  - The Unified Process requires the project team to focus on the most critical risks early in the project life cycle.
  - The deliverables of each iteration, especially in the Elaboration phase, must be selected in order to ensure that the greatest risks are addressed first.

- **Introduction to UML**
- UML is a **language** used for object-oriented analysis and design.
- UML includes a set of graphical notation techniques to create visual models of software systems.
- Three developers of UML are Grady Booch, Ivar Jacobson and James Rumbaugh.
- UML is a language for visualizing, specifying, constructing, documenting.
- UML is **not a development method**. It is designed to be compatible with the object-oriented software development methods.

---

## 2.7 VIEWS IN UML

---

### 1. Functional View:

- This view describes **functional requirements** of the system.
- Use case diagrams give **static functional view** for functions and static relationships.
- Activity diagrams give **dynamic functional view**.

### 2. Static Structural View:

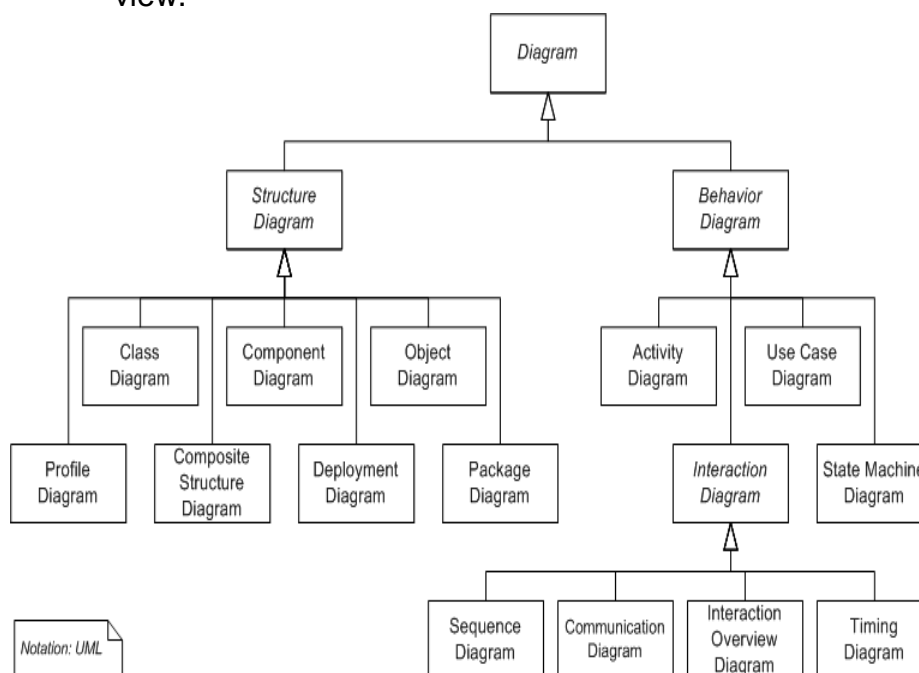
The **class and object diagrams** give the structural view of system.

### 3. Behavioral (dynamic structural) view:

- **Interaction diagrams**- collaboration diagrams and sequence diagrams describe sequences of interactions between objects.
- **State transition diagrams** show state-based behavior of objects.

### 4. Architectural View:

- This view describes logical and physical structure.
- Component diagrams and deployment diagrams are used in this view.



---

## 2.8 UML DIAGRAMS

---

Diagrams are the heart of UML. These diagrams are broadly categorized as structural and behavioral diagrams.

- Structural diagrams are consists of static diagrams like class diagram, object diagram etc.
- Behavioral diagrams are consists of dynamic diagrams like sequence diagram, collaboration diagram etc.

The static and dynamic nature of a system is visualized by using these diagrams.

### **Class diagrams:**

Class diagrams are the most popular UML diagrams used by the object oriented community. It describes the objects in a system and their relationships. Class diagram consists of attributes and functions.

A single class diagram describes a specific aspect of the system and the collection of class diagrams represents the whole system. Basically the class diagram represents the static view of a system.

Class diagrams are the only UML diagrams which can be mapped directly with object oriented languages. So it is widely used by the developer community.

### **Object Diagram:**

An object diagram is an instance of a class diagram. So the basic elements are similar to a class diagram. Object diagrams are consists of objects and links. It captures the instance of the system at a particular moment.

Object diagrams are used for prototyping, reverse engineering and modeling practical scenarios.

### **Component Diagram:**

Component diagrams are special kind of UML diagram to describe static implementation view of a system. Component diagrams consist of physical components like libraries, files, folders etc.

This diagram is used from implementation perspective. More than one component diagrams are used to represent the entire system. Forward and reverse engineering techniques are used to make executables from component diagrams.

### **Deployment Diagram:**

Component diagrams are used to describe the static deployment view of a system. These diagrams are mainly used by system engineers.

Deployment diagrams are consists of nodes and their relationships. An efficient deployment diagram is an integral part of software application development.

**Use Case Diagram;**

Use case diagram is used to capture the dynamic nature of a system. It consists of use cases, actors and their relationships. Use case diagram is used at a high level design to capture the requirements of a system.

So it represents the system functionalities and their flow. Although the use case diagrams are not a good candidate for forward and reverse engineering but still they are used in a slightly differently way to model it.

**Interaction Diagram:**

Interaction diagrams are used for capturing dynamic nature of a system. Sequence and collaboration diagrams are the interaction diagrams used for this purpose.

Sequence diagrams are used to capture time ordering of message flow and collaboration diagrams are used to understand the structural organization of the system. Generally a set of sequence and collaboration diagrams are used to model an entire system.

**Statechart Diagram:**

Statechart diagrams are one of the five diagrams used for modeling dynamic nature of a system. These diagrams are used to model the entire life cycle of an object. Activity diagram is a special kind of Statechart diagram.

State of an object is defined as the condition where an object resides for a particular time and the object again moves to other states when some events occur. Statechart diagrams are also used for forward and reverse engineering.

**Activity Diagram:**

Activity diagram is another important diagram to describe dynamic behaviour. Activity diagram consists of activities, links, relationships etc. It models all types of flows like parallel, single, concurrent etc.

Activity diagram describes the flow control from one activity to another without any messages. These diagrams are used to model high level view of business requirements.





## USE CASE DIAGRAMS

### Unit Structure

- 3.1 Introduction
- 3.2 Requirement capture with use case
- 3.3 Building blocks of use case diagram
- 3.4 Actors
- 3.5 Dependencies
- 3.6 Generalizations

---

### 3.1 INTRODUCTION

---

To model a system the most important aspect is to capture the dynamic behaviour. To clarify a bit in details, *dynamic behaviour* means the behaviour of the system when it is running /operating.

So only static behaviour is not sufficient to model a system rather dynamic behaviour is more important than static behaviour. In UML there are five diagrams available to model dynamic nature and use case diagram is one of them. Now as we have to discuss that the use case diagram is dynamic in nature there should be some internal or external factors for making the interaction.

These internal and external agents are known as actors. So use case diagrams are consists of actors, use cases and their relationships. The diagram is used to model the system/subsystem of an application. A single use case diagram captures a particular functionality of a system.

So to model the entire system numbers of use case diagrams are used.

---

### 3.2 REQUIREMENT CAPTURE WITH USE CASE

---

The purpose of use case diagram is to capture the dynamic aspect of a system. But this definition is too generic to describe the purpose.

Because other four diagrams (activity, sequence, collaboration and Statechart) are also having the same purpose. So we will look into some specific purpose which will distinguish it from other four diagrams.

Use case diagrams are used to gather the requirements of a system including internal and external influences. These requirements are mostly design requirements. So when a system is analyzed to gather its functionalities use cases are prepared and actors are identified.

Now when the initial task is complete use case diagrams are modelled to present the outside view.

So in brief, the purposes of use case diagrams can be as follows:

- Used to gather requirements of a system.
- Used to get an outside view of a system.
- Identify external and internal factors influencing the system.
- Show the interacting among the requirements are actors.

### 3.3 BUILDING BLOCKS OF USE CASE DIAGRAM

As UML describes the real time systems it is very important to make a conceptual model and then proceed gradually. Conceptual model of UML can be mastered by learning the following three major elements:

UML building blocks

Rules to connect the building blocks

Common mechanisms of UML

This chapter describes all the UML building blocks. The building blocks of UML can be defined as:

Things

Relationships

Diagrams

#### (1) Things:

Things are the most important building blocks of UML. Things can be:

Structural

Behavioral

Grouping

Annotational

#### Structural things:

The Structural things define the static part of the model. They represent physical and conceptual elements. Following are the brief descriptions of the structural things.

#### Class:

Class represents set of objects having similar responsibilities.



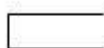
#### Interface:

Interface defines a set of operations which specify the responsibility of a class.



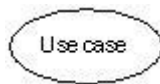
#### Collaboration:

Collaboration defines interaction between elements.

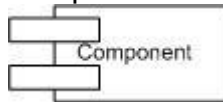


#### Use case:

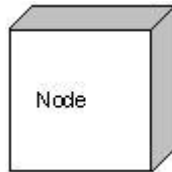
Use case represents a set of actions performed by a system for a specific goal.

**Component:**

Component describes physical part of a system.

**Node:**

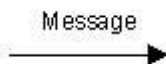
A node can be defined as a physical element that exists at run time.

**Behavioral things:**

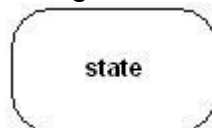
A behavioral thing consists of the dynamic parts of UML models. Following are the behavioral things:

**Interaction:**

Interaction is defined as a behavior that consists of a group of messages exchanged among elements to accomplish a specific task.

**State machine:**

State machine is useful when the state of an object in its life cycle is important. It defines the sequence of states an object goes through in response to events. Events are external factors responsible for state change.

**Grouping things:**

Grouping things can be defined as a mechanism to group elements of a UML model together. There is only one grouping thing available:

**Package:**

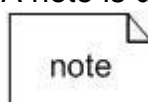
Package is the only one grouping thing available for gathering structural and behavioral things.

**Annotational things:**

Annotational things can be defined as a mechanism to capture remarks, descriptions, and comments of UML model elements. Note is the only one Annotational thing available.

**Note:**

A note is used to render comments, constraints etc of an UML element.



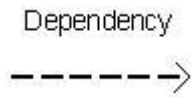
## (2) Relationship :

Relationship is another most important building block of UML. It shows how elements are associated with each other and this association describes the functionality of an application.

There are four kinds of relationships available.

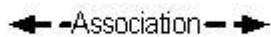
### Dependency:

Dependency is a relationship between two things in which change in one element also affects the other one.



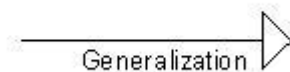
### Association:

Association is basically a set of links that connects elements of an UML model. It also describes how many objects are taking part in that relationship.



### Generalization:

Generalization can be defined as a relationship which connects a specialized element with a generalized element. It basically describes inheritance relationship in the world of objects.



### Realization:

Realization can be defined as a relationship in which two elements are connected. One element describes some responsibility which is not implemented and the other one implements them. This relationship exists in case of interfaces.



## (3) UML Diagrams:

UML diagrams are the ultimate output of the entire discussion. All the elements, relationships are used to make a complete UML diagram and the diagram represents a system.

The visual effect of the UML diagram is the most important part of the entire process. All the other elements are used to make it a complete one.

UML includes the following nine diagrams and the details are described in the following chapters.

1. Class diagram
2. Object diagram
3. Use case diagram
4. Sequence diagram
5. Collaboration diagram
6. Activity diagram
7. Statechart diagram
8. Deployment diagram
9. Component diagram

We would discuss all these diagrams in subsequent chapters of this book.

### 3.4 ACTORS

An actor is a user or external system with which a system being modeled interacts. For example, our project management system involves various types of users, including project managers, resource managers, human resources, and system administrators. These users are all actors.

An actor is external to a system, interacts with the system, may be a human user or another system, and has goals and responsibilities to satisfy in interacting with the system. Actors address the question of who and what interacts with a system. In the UML, an actor is shown as a "stick figure" icon, or as a class marked with the actor keyword and labeled with the name of the actor class.

Following figure shows various actors associated with the project management system:

#### ***A project manager***

Responsible for ensuring that a project delivers a quality product within specified time and cost, and within specified resource constraints

#### ***A resource manager***

Responsible for ensuring that trained and skilled human resources are available for projects

#### ***A human resource***

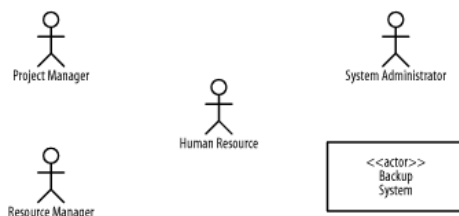
Responsible for ensuring that worker skills are maintained, and that quality work is completed for a project

#### ***A system administrator***

Responsible for ensuring that a project management system is available for a project

#### ***A backup system***

Responsible for housing backup data for a project management system



### Use case guidelines for use case models

Use case diagrams are considered for high level requirement analysis of a system. So when the requirements of a system are analyzed the functionalities are captured in use cases.

So we can say that uses cases are nothing but the system functionalities written in an organized manner. Now the second things

which are relevant to the use cases are the actors. Actors can be defined as something that interacts with the system.

The actors can be human user, some internal applications or may be some external applications. So in a brief when we are planning to draw an use case diagram we should have the following items identified.

- Functionalities to be represented as an use case
- Actors
- Relationships among the use cases and actors.

Use case diagrams are drawn to capture the functional requirements of a system. So after identifying the above items we have to follow the following guidelines to draw an efficient use case diagram.

- The name of a use case is very important. So the name should be chosen in such a way so that it can identify the functionalities performed.
- Give a suitable name for actors.
- Show relationships and dependencies clearly in the diagram.
- Do not try to include all types of relationships. Because the main purpose of the diagram is to identify requirements.
- Use note when ever required to clarify some important points.

The following is a sample use case diagram representing the order management system. So if we look into the diagram then we will find three use cases (Order, SpecialOrder and NormalOrder) and one actor which is customer.

The *SpecialOrder* and *NormalOrder* use cases are extended from *Order* use case. So they have extends relationship. Another important point is to identify the system boundary which is shown in the picture. The actor *Customer* lies outside the system as it is an external user of the system.

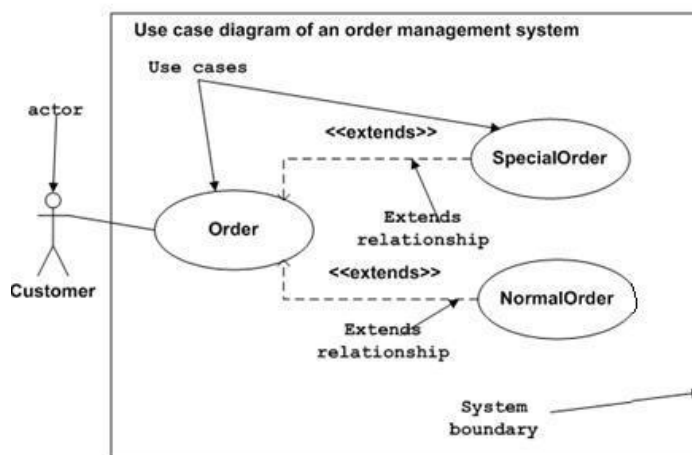


Figure: Sample Use Case diagram

### 3.5 DEPENDENCIES

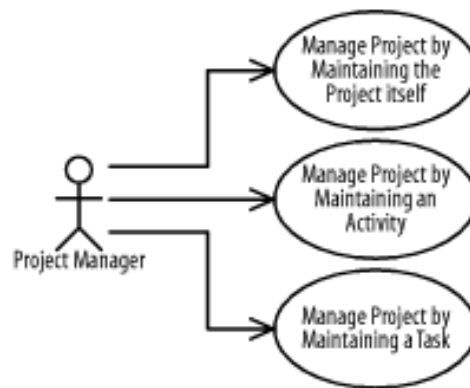
A model may have many use cases, so how do we organize the use cases that define what a system should do? And how do we use this information about use cases to determine how best to execute a project while considering how use cases are related to one another, including what some use cases might have in common, and also taking into account use cases that are options of other use cases? Specialized types of dependencies, called include and extend dependencies,

address these questions; The next few sections discuss these specialized types of dependencies.

### Include Dependencies

Perhaps we wish to log the activities of project managers, resources managers, and system administrators as they interact with the project management system. Figure A below elaborates on the use cases and figure B show that the activities of the project manager, resource managers, and system administrators are logged when they are performing the use cases shown in the diagram. Thus, logging activities are common to these three use cases. We can use an include dependency to address this type of situation by factoring out and reusing common behavior from multiple use cases.

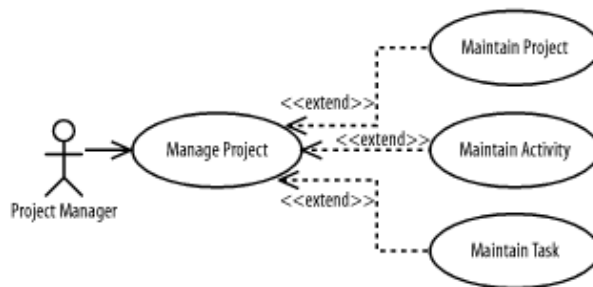
**Figure A. Use cases with common behavior**



An include dependency from one use case (called the base use case) to another use case (called the inclusion use case) indicates that the base use case will include or call the inclusion use case. A use case may include multiple use cases, and it may be included in multiple use cases. An include dependency is shown as a dashed arrow from the base use case to the inclusion use case marked with the include keyword. The base use case is responsible for identifying where in its behavior sequence or at which step to include the inclusion use case. This identification is not done in the UML diagram, but rather in the textual description of the base use case.

B refines A using include dependencies. The Log Activity use case is common to the Manage Project, Manage Resource, and Administer System use cases, so it is factored out and included by these use cases.

**Figure B. Include dependencies**

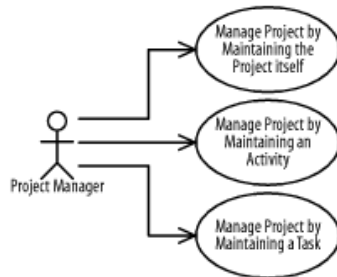


You can use an include dependency when a use case may be common to multiple other use cases and is therefore factored out of the different use cases so that it may be reused. The Log Activity use case in B is included in the Manage Project, Manage Resource, and Administer System use cases. Consequently, you must analyze and develop that use case before you develop the three use cases that depend on it.

## Extend Dependencies

Projects are made of activities, and activities are made of tasks. Figure C elaborates the Manage Project use case and shows that a project manager may manage projects by maintaining the project itself, its activities, or its tasks. Thus, maintaining the project, its activities, and its tasks are options of managing a project. You can use an extend dependency to address this situation by factoring out optional behavior from a use case.

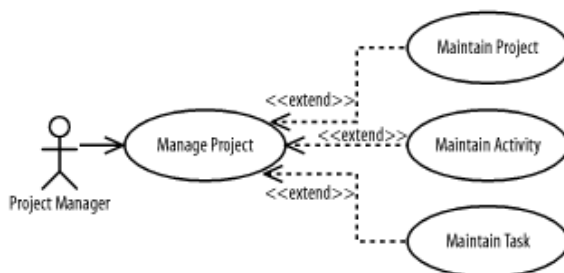
**Figure C. Use cases with optional behavior**



An extend dependency from one use case (called the extension use case) to another use case (called the base use case) indicates that the extension use case will extend (or be inserted into) and augment the base use case. A use case may extend multiple use cases, and a use case may be extended by multiple use cases. An extend dependency is shown as a dashed arrow from the extension use case to the base use case marked with the extend keyword. The base use case is responsible for identifying at which steps in its behavior sequence the extending use cases may be inserted.

Figure D refines Figure C using extend dependencies. The Maintain Project, Maintain Activity, and Maintain Task use cases are options of the Manage Project use case, so Manage Project is factored out and extends those three use cases.

**Figure D. Simple extend dependencies**



The location in a base use case at which another behavior sequence may be inserted is called an extension point. Extension points for a use case may be listed in a compartment labeled "Extension Points" where each extension point is shown inside the compartment with an extension-point name followed by a colon followed by a suitable description of the location of the extension point in the use case's



behavior sequence. Locations may be described as being before, after, or in-the-place-of a step in the base use case's behavior sequence. For example, the Manage Project use case may have a behavior sequence for finding a project on which to work followed by an extension point named Project Selected followed by another behavior. The Project Selected extension point may be described as occurring after a project is found but before it is actually worked on.

An extend dependency is responsible for defining when an extension use case is inserted into the base use case by specifying a condition that must be satisfied for the insertion to occur. The condition may be shown following the extend keyword enclosed within square brackets followed by the extension point name enclosed in parentheses. For example, other use cases may be inserted into the Project Selected extension point just described for the Manage Project use case. Such behavior sequences may include reviewing and updating project information, or selecting a specific version of a project before managing the details of the project in the succeeding behavior sequences.

---

### 3.5 GENERALIZATIONS

---

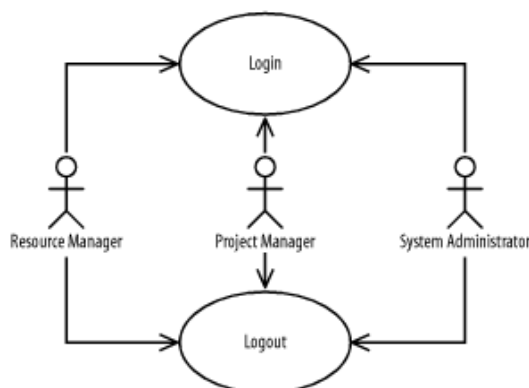
Actors may be similar in how they use a system; for example, project managers, resource managers, and system administrators may log in and out of our project management system. Use cases may be similar in the functionality provided to users; for example, a project manager may publish a project's status in two ways: by generating a report to a printer or by generating a web site on a project web server.

Given that there may be similarities between actors and use cases, how do we organize the use cases that define what a system should do? And how do we use the information about similarities between actors and use cases to determine how best to execute a project? Specialized types of generalizations, called actor and use case generalizations, address these questions.

#### Actor Generalizations

Figure E shows that project managers, resource managers, and system administrators may log in and out of the project management system. Thus, logging in and out is common to these actors. Actor generalizations address such situations by factoring out and reusing similarities between actors.

Figure E

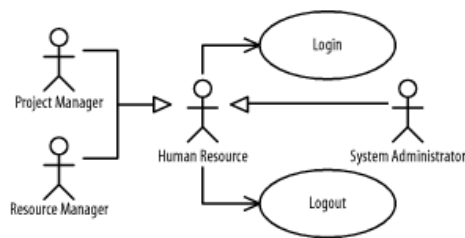


An actor generalization from a more specific, or specialized, actor to a more general, or generalized, actor indicates that instances of the more specific actor may be substituted for instances of the more general

actor. An actor may specialize multiple actors, and an actor may be specialized by multiple actors. An actor generalization between actors is shown as a solid-line path from the more specific actor to the more general actor, with a large hollow triangle at the end of the path connected to the more general actor.

Figure F refines Figure E using actor generalizations between actors. A human resource initiates the Login and Logout use cases. Project managers, resource managers, and system administrators are human resources.

Figure F



Use an actor generalization between actors when one actor is similar to another, but has specific interactions in which it participates or initiates. For example, any human resource may log in and out, but project managers, resources managers, and system administrators make more specialized use of the project management system. Because the Project Manager, Resource Manager, and System Administrator actors are specialized Human Resource actors, they benefit from the use cases in which the Human Resource actor is involved. Therefore, by developing the Login and Logout use cases, we provide the functionality described by those use cases for all the actors of our system.

## **Use-Case Generalizations**

Figure G shows that a project manager may publish a project's status in two ways: by generating a report to a printer or by generating a web site on a project web server. Thus, publishing a project's status and all the processing involved in collecting and preparing the data for publication is common to these use cases. You can use a use-case generalization to address this situation by factoring out and reusing similar behavior from multiple use cases.

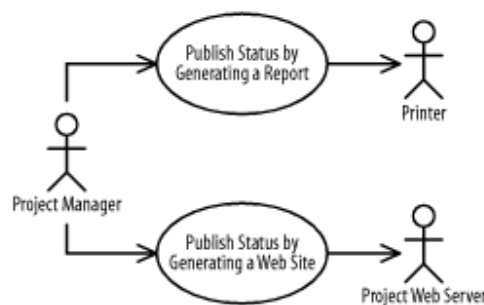


Figure G

A use-case generalization from a more specific, or specialized, use case to a more general, or generalized, use case indicates that the more specific use case receives or inherits the actors, behavior sequences, and extension points of the more general use case, and that instances of the more specific use case may be substituted for instances of the more general use case. The specific use case may include other

actors, define new behavior sequences and extension points, and modify or specialize the behavior sequences it receives or inherits. A use case may specialize multiple use cases, and a use case may be specialized by multiple use cases. A use-case generalization between use cases is shown as a solid-line path from the more specific use case to the more general use case, with a large hollow triangle at the end of the path connected to the more general use case.



# 4

## ACTIVITY DIAGRAM

### Unit Structure

- 4.1 Introduction
- 4.2 Overview
- 4.3 Elements of an Activity diagram
- 4.4 Action State\ Activity state
- 4.5 Object Node
- 4.6 Control Node
- 4.7 How to draw Activity Diagram?

---

### 4.1 INTRODUCTION

---

This chapter focuses on activity diagrams, which depict the activities and responsibilities of elements that make up a system. First, we introduce activity diagrams and how they are used. Next, we discuss action states, activity state and their details. Finally, we go over control, flows and their details. we include suggestions relating to activity diagrams.

Activity modeling is a specialized type of behavioral modeling concerned with modeling the activities and responsibilities of elements. You usually apply activity modeling in conjunction with sequence and collaboration modeling to explore the activities and responsibilities of interacting and collaborating elements.

---

### 4.2 OVERVIEW

---

Activity diagram is another important diagram in UML to describe dynamic aspects of the system.

Activity diagram is basically a flow chart to represent the flow from one activity to another activity. The activity can be described as an operation of the system.

So the control flow is drawn from one operation to another. This flow can be sequential, branched or concurrent. Activity diagrams deals with all type of flow control by using different elements like fork, join etc.

The basic purposes of activity diagrams are similar to other four diagrams. It captures the dynamic behavior of the system.

Other four diagrams are used to show the message flow from one object to another but activity diagram is used to show message flow from one activity to another.

Activity is a particular operation of the system. Activity diagrams are not only used for visualizing dynamic nature of a system but they are also used to construct the executable system by using forward and reverse engineering techniques. The only missing thing in activity diagram is the message part.

It does not show any message flow from one activity to another. Activity diagram is some time considered as the flow chart. Although the diagrams looks like a flow chart but it is not. It shows different flow like parallel, branched, concurrent and single.

So the purposes can be described as:


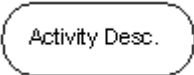
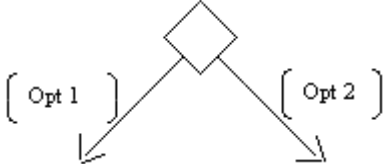
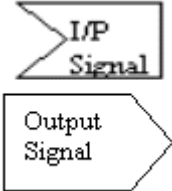
Draw the activity flow of a system.

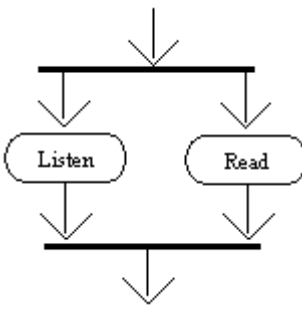

Describe the sequence from one activity to another.

Describe the parallel, branched and concurrent flow of the system.

### 4.3 ELEMENTS OF AN ACTIVITY DIAGRAM

An Activity diagram consists of the following behavioral elements:

Element and its description	Symbol
<b>Initial Activity:</b> This shows the starting point or first activity of the flow. Denoted by a solid circle. This is similar to the notation used for Initial State.	
<b>Activity:</b> Represented by a rectangle with rounded (almost oval) edges.	
<b>Decisions:</b> Similar to flowcharts, a logic where a decision is to be made is depicted by a diamond, with the options written on either sides of the arrows emerging from the diamond, within box brackets.	
<b>Signal:</b> When an activity sends or receives a message, that activity is called a signal. Signals are of two types: Input signal (Message receiving activity) shown by a concave polygon and Output signal (Message sending activity) shown by a convex polygon.	

<p><b>Concurrent Activities:</b> Some activities occur simultaneously or in parallel. Such activities are called concurrent activities. For example, listening to the lecturer and looking at the blackboard is a parallel activity. This is represented by a horizontal split (thick dark line) and the two concurrent activities next to each other, and the horizontal line again to show the end of the parallel activity.</p>	
<p><b>Final Activity:</b> The end of the Activity diagram is shown by a bull's eye symbol, also called as a final activity</p>	

## 4.4 ACTION STATE\ ACTIVITY STATE

As elements communicate with one another within a society of objects, each element has the responsibility of appropriately reacting to the communications it receives. An action state represents processing as an element fulfills a responsibility. There are various types of action states, including simple, initial, and final action states. The next few sections discuss these different types of action states.

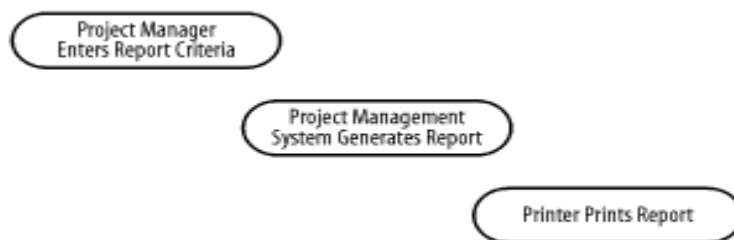
### Simple Action States

A simple action state represents processing. For example, the project management system may have the following simple action states:

- **Project Manager Enters Report Criteria** : Indicates that the project manager enters report criteria
- **Project Management System Generates Report** : Indicates that the project management system generates a report
- **Printer Prints Report** : Indicates that the printer prints the report

In the UML, an action state is shown as a shape with a straight top and bottom and convex arcs on the two sides, and is labeled with the name of an operation or a description of the processing. Figure A shows the various action states associated with the project management system.

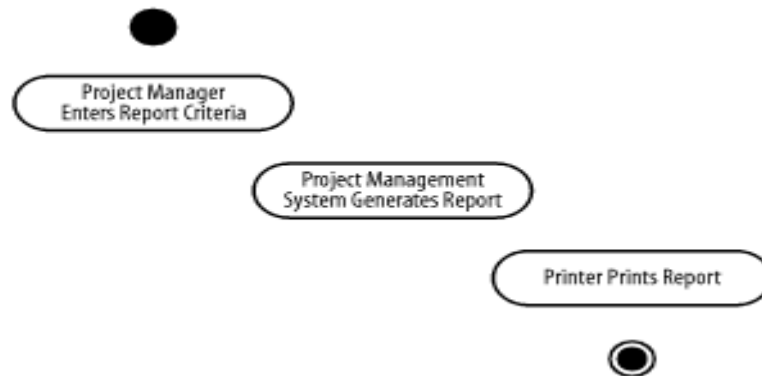
Figure A. Simple action states



## Initial and Final Action States

An initial action state indicates the first action state on an activity diagram. In the UML, an initial action state is shown using a small solid filled circle. A final action state indicates the last action state on an activity diagram. In the UML, a final action state is shown using a circle surrounding a small solid filled circle (a bull's eye). Figure B updates Figure A with an initial and final action state. An activity diagram may have only one initial action state, but may have any number of final action states.

Figure B. Simple, initial, and final action states



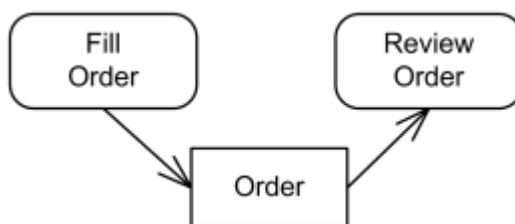

---

## 4.5 OBJECT NODE

---

An **object** node is an activity node that is part of defining object flow in an activity. It indicates that an instance of a particular Classifier, possibly in a particular state, may be available at a particular point in the activity. Object nodes can be used in a variety of ways, depending on where objects are flowing from and to.

Object nodes are notated as rectangles. A name labeling the node is placed inside the symbol, where the name indicates the type of the object node, or the name and type of the node in the format "name:type."



### Object flow of Orders between Fill Order and Review Order actions

The name can also be qualified by a state or states, which is to be written within brackets below the name of the type. Upper bounds, ordering, and control type other than the defaults are notated in braces underneath the object node.

## 4.6 CONTROL NODE

Control node is an activity node used to coordinate the flows between other nodes. It includes:

- **Initial Node**

Initial node is a control node at which flow starts when the activity is invoked.

A control token is placed at the initial node when the activity starts, but not in initial nodes in structured nodes contained by the activity. Tokens in an initial node are offered to all outgoing edges. For convenience, initial nodes are an exception to the rule that control nodes cannot hold tokens if they are blocked from moving downstream, for example, by guards.

Activity may have more than one initial node. In this case, invoking the activity starts multiple flows, one at each initial node.

Note that flows can also start at other nodes, so initial nodes are not required for an activity to start execution.

Initial nodes are shown as a small solid circle.

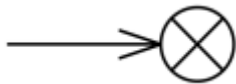


Activity initial node.

- **Flow Final Node**

Flow final node is a control final node that terminates a flow. It destroys all tokens that arrive at it but has no effect on other flows in the activity. Flow final was introduced in UML 2.0.

The notation for flow final node is small circle with X inside.



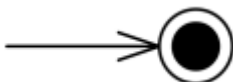
Flow final node.

- **Activity Final Node**

Activity final node is a control final node that stops all flows in an activity. Activity final was introduced in UML 2.0.

An activity may have more than one activity final node. The first one reached stops all flows in the activity. A token reaching an activity final node terminates the activity. In particular, it stops all executing actions in the activity, and destroys all tokens in object nodes, except in the output activity parameter nodes. Terminating the execution of synchronous invocation actions also terminates whatever behaviors they are waiting on for return. Any behaviors invoked asynchronously by the activity are not affected. If it is not desired to abort all flows in the activity, use flow final instead.

Activity final nodes are shown as a solid circle with a hollow circle inside. It can be thought of as a goal notated as "bull's eye," or target.



Activity final node.

- **Decision Node**

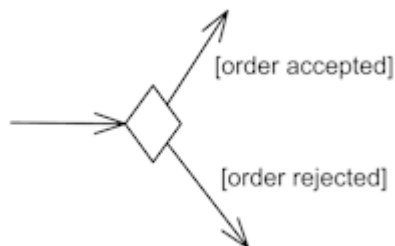
Decision node is a control node that accepts tokens on one or two incoming edges and selects one outgoing edge from one or more outgoing flows. Decision nodes were introduced in UML to support conditionals in activities.

The edges coming into and out of a decision node, other than the decision input flow (if any), must be either all object flows or all control flows.

Each token arriving at a decision node can traverse only one outgoing edge. Tokens are not duplicated. Each token offered by the incoming edge is offered to the outgoing edges.

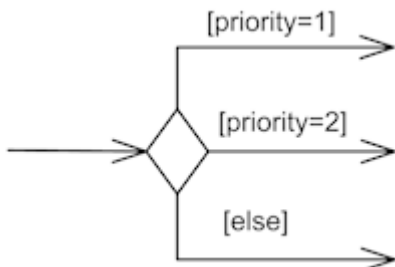
Which of the edges is actually traversed depends on the evaluation of the guards on the outgoing edges. The order in which guards are evaluated is not defined, i.e. we should not rely on any visual or text description order.

The notation for a decision node is a diamond-shaped symbol.



Decision node with two outgoing edges with guards.

The modeler should arrange that each token only be chosen to traverse one outgoing edge. For decision points, a predefined guard "else" may be defined for at most one outgoing edge.



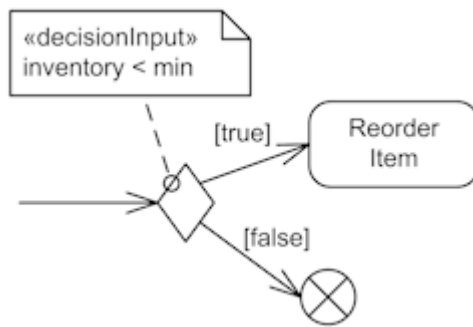
Decision node with three outgoing edges and [else] guard.

Decision can have decision input behavior specified. Decision input behaviors were introduced in UML to avoid redundant recalculations in guards.

In this case each data token is passed to the behavior before guards are evaluated on the outgoing edges. The behavior is invoked without input for control tokens. The output of the behavior is available to each guard. Because the behavior is used during the process of offering tokens to outgoing edges, it may be run many times on the same token before the token is accepted by those edges. This means the behavior cannot have side effects.

Decision input behavior is specified by the keyword «decisionInput» and some decision behavior or condition placed in a note symbol, and attached to the appropriate decision node.

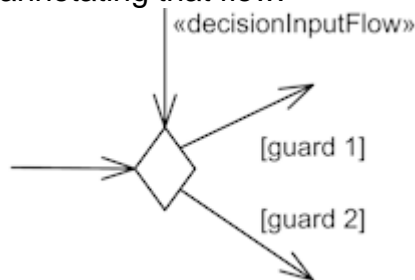




Decision node with decision input behavior.

Decision may also have decision input flow. In this case the tokens offered on the decision input flow that are made available to the guard on each outgoing edge determine whether the offer on the regular incoming edge is passed along that outgoing edge.

A decision input flow is specified by the keyword «decisionInputFlow» annotating that flow.



Decision node with decision input flow.

If there are both a decision input behavior as well as decision input flow, the token offered on the decision input flow is passed to the behavior (as the only argument if the regular incoming edge is control flow, as the second argument if it is an object flow). Decision nodes with the additional decision input flow offer tokens to outgoing edges only when one token is offered on each incoming edge.

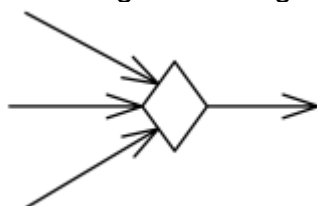
### • Merge Node

Merge node is a control node that brings together multiple incoming alternate flows to accept single outgoing flow. There is no joining of tokens. Merge should not be used to synchronize concurrent flows.

For example, if a decision is used after a fork, the two flows coming out of the decision need to be merged into one before going to a join; otherwise, the join will wait for both flows, only one of which will arrive.

All edges coming into and out of a merge node must be either object flows or control flows.

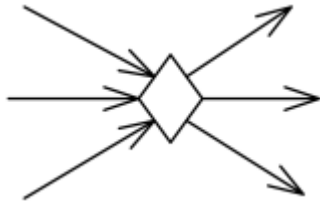
The notation for a merge node is a diamond-shaped symbol with two or more edges entering it and a single activity edge leaving it.



Merge node with three incoming edges and a single outgoing edge

The functionality of merge node and decision node can be combined by using the same node symbol, as illustrated below. This

case maps to a model containing a merge node with all the incoming edges shown in the diagram and one outgoing edge to a decision node that has all the outgoing edges shown in the diagram.



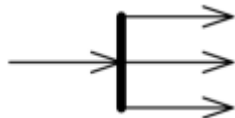
Merge node and decision node combined using the same symbol

- **Fork Node**

Fork node is a control node that has one incoming edge and multiple outgoing edges and is used to split incoming flow into multiple concurrent flows. Fork nodes are introduced to support parallelism in activities. As compared to UML 1.5, UML 2.0 activity forks model unrestricted parallelism.

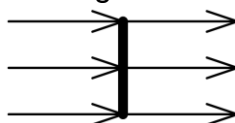
Tokens arriving at a fork are duplicated across the outgoing edges. If at least one outgoing edge accepts the token, duplicates of the token are made and one copy traverses each edge that accepts the token. The outgoing edges that did not accept the token due to failure of their targets to accept it, keep their copy in an implicit FIFO queue until it can be accepted by the target. The rest of the outgoing edges do not receive a token.

The notation for a fork node is a line segment with a single activity edge entering it, and two or more edges leaving it.



Fork node with a single activity edge entering it, and three edges leaving it.

The functionality of join node and fork node can be combined by using the same node symbol. This case maps to a model containing a join node with all the incoming edges shown in the diagram and one outgoing edge to a fork node that has all the outgoing edges shown in the diagram.



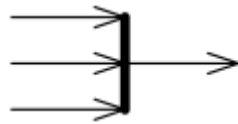
Combined join node and fork node.

If guards are used on edges outgoing from forks, the modelers should ensure that no downstream joins depend on the arrival of tokens passing through the guarded edge. If that cannot be avoided, then a decision node should be introduced to have the guard, and shunt the token to the downstream join if the guard fails.

- **Join Node**

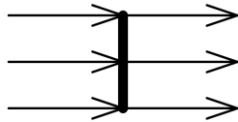
Join node is a control node that has multiple incoming edges and one outgoing edge and is used to synchronize incoming concurrent flows. Join nodes are introduced to support parallelism in activities.

The notation for a join node is a line segment with several activity edges entering it, and only one edge leaving it.



Join node with three activity edges entering it, and a single edge leaving it.

The functionality of join node and fork node can be combined by using the same node symbol. This case maps to a model containing a join node with all the incoming edges shown in the diagram and one outgoing edge to a fork node that has all the outgoing edges shown in the diagram.



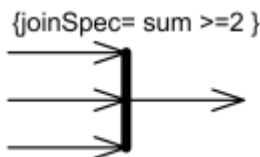
Combined join node and fork node.

Join nodes have a join specification which is Boolean value specification using the names of the incoming edges to specify the conditions under which the join will emit a token.

The join specification is evaluated whenever a new token is offered on any incoming edge. The evaluation is not interrupted by any new tokens offered during the evaluation, nor are concurrent evaluations started when new tokens are offered during an evaluation.

The default join specification is the reserved string "and". It is equivalent to a specification that requires at least one token offered on each incoming edge.

Join specifications are shown in curly braces near the join node as joinSpec=....



Join node with join specification shown in curly braces.

---

## 4.7 HOW TO DRAW ACTIVITY DIAGRAM?

---

Activity diagrams are mainly used as a flow chart consists of activities performed by the system. But activity diagram are not exactly a flow chart as they have some additional capabilities. These additional capabilities include branching, parallel flow, swimlane etc.

Before drawing an activity diagram we must have a clear understanding about the elements used in activity diagram. The main element of an activity diagram is the activity itself. An activity is a function performed by the system. After identifying the activities we need to understand how they are associated with constraints and conditions.

So before drawing an activity diagram we should identify the following elements:

- Activities
- Association
- Conditions
- Constraints

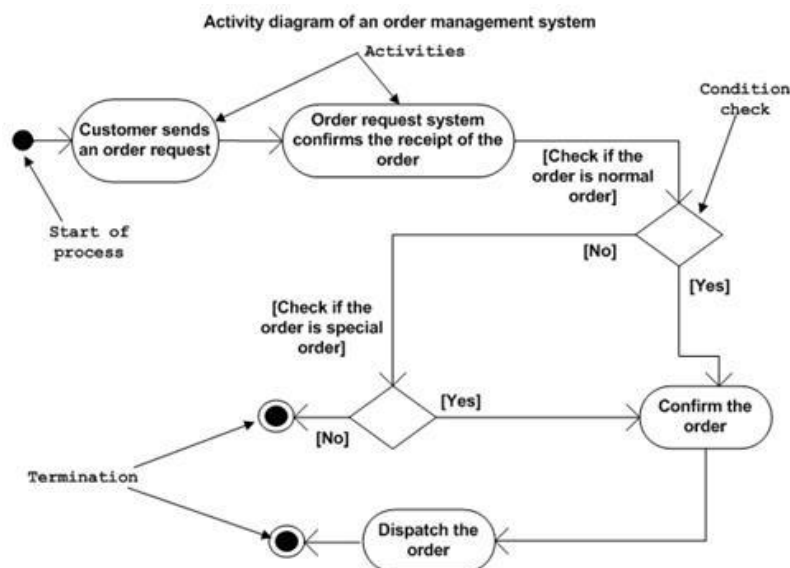
Once the above mentioned parameters are identified we need to make a mental layout of the entire flow. This mental layout is then transformed into an activity diagram.

e.g. The following is an example of an activity diagram for order management system. In the diagram four activities are identified which are associated with conditions. One important point should be clearly understood that an activity diagram cannot be exactly matched with the code. The activity diagram is made to understand the flow of activities and mainly used by the business users.

The following diagram is drawn with the four main activities:

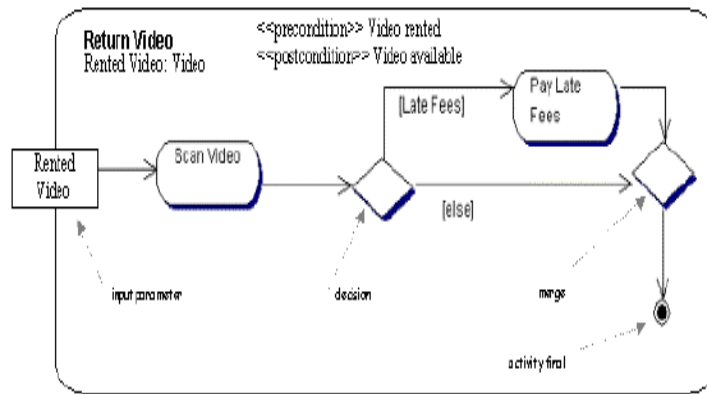
- Send order by the customer
- Receipt of the order
- Confirm order
- Dispatch order

After receiving the order request condition checks are performed to check if it is normal or special order. After the type of order is identified dispatch activity is performed and that is marked as the termination of the process.



### Action decomposition (rake)

Often, an action can be decomposed into a sequence of other actions. As a result, the action is implemented by a subactivity. When we implement an action via another activity, we place a rake in the node that represents the action. This rake indicates that a subactivity diagram implements the action.



### Partition Swimlanes:

As discussed earlier in this chapter, a swimlane is a visual region in an activity diagram that indicates the element that has responsibility for action states within the region. For example, the project management system may have the following swimlanes, which are illustrated in following figure

#### Project Manager

Shows the action states that are the responsibility of a project manager. The swimlane makes it obvious that the project manager is responsible for entering data, thus the rather cumbersome action state name of Project Manager Enters Data may be shortened to Enter Data.

#### Project Management System

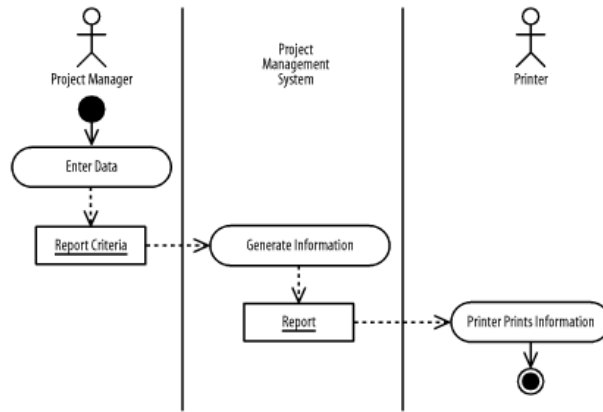
Shows the action states that are the responsibility of the project management system. Again, because the swimlane makes it obvious who (or what, in this case) is generating information, the rather cumbersome action state name of Project Management System Generates Information may be shortened to Generate Information.

#### Printer

Shows the action states that are the responsibility of a printer. Because of this swimlane, the rather cumbersome action state name of Printer Prints Information may be shortened to Print Information.

Notice how the use of swimlanes allows me to rename the action states to omit the responsible element for each action state.

In the UML, a swimlane is shown as a visual region separated from neighboring swimlanes by vertical solid lines on both sides and labeled at the top with the element responsible for action states within the swimlane. Following figure shows the swimlanes associated with the project management system.



# 5

## CLASS DIAGRAMS

### Unit Structure

- 5.1 Introduction
- 5.2 Class
- 5.3 Attributes and operations/methods
- 5.4** Responsibilities of Classes
- 5.5 Abstract Class
- 5.6 Standard Class Stereotypes
- 5.7 Dependency

---

### 5.1 INTRODUCTION

---

Class diagram is UML structure diagram which shows structure of the designed system at the level of classes and interfaces, shows their features, constraints and relationships - associations, generalizations, dependencies, etc.

---

### 5.2 CLASS

---

A **class** is a classifier which describes a set of objects that share the same

- features
- constraints
- semantics (meaning).

### Class Diagrams

UML class diagrams model static class relationships that represent the fundamental architecture of the system. Note that these diagrams describe the relationships between *classes*, not those between specific *objects* instantiated from those classes. Thus the diagram applies to *all the objects* in the system.

A class diagram consists of the following features:

- **Classes:** These titled boxes represent the classes in the system and contain information about the name of the class, fields,

methods and access specifiers. Abstract roles of the class in the system can also be indicated.

- *Interfaces*: These titled boxes represent interfaces in the system and contain information about the name of the interface and its methods.

A class is shown as a solid-outline rectangle containing the class name, and optionally with compartments separated by horizontal lines containing features or other members of the classifier.

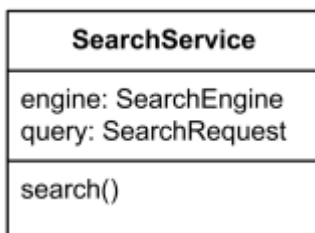
As class is the most widely used classifier, there is no need to add the "class" keyword in guillemets above the class name. Class name should be **centered** and in **bold** face, with the first letter of class name capitalized (if the character set supports upper case).



**Class** Customer - details suppressed

### 5.3 ATTRIBUTES AND OPERATIONS/METHODS

Features of a class are **attributes** and **operations**. When class is shown with three **compartments**, the middle compartment holds a list of **attributes** and the bottom compartment holds a list of **operations**. Attributes and operations should be **left justified** in **plain** face, with the first letter of the names in **lower** case.



**Class** SearchService - analysis level details

Characteristics represented by feature may be of the classifier's instances considered individually (**not static**) or of the classifier itself (**static**). The same feature cannot be static in one context and non static in another.

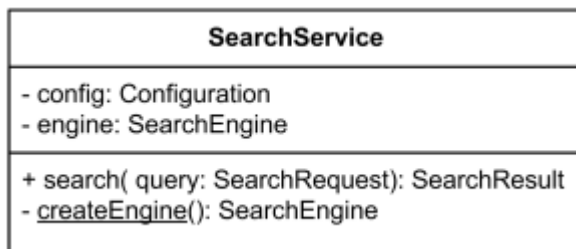
With regard to **static features**, two alternative semantics are recognized. Static feature may have:

- different values for different featuring classifiers, or
- the same value for all featuring classifiers.

Static features are **underlined** - but only the names. An ellipsis (...) as the final element of a list of features indicates that additional features exist but are **not shown** in that list.

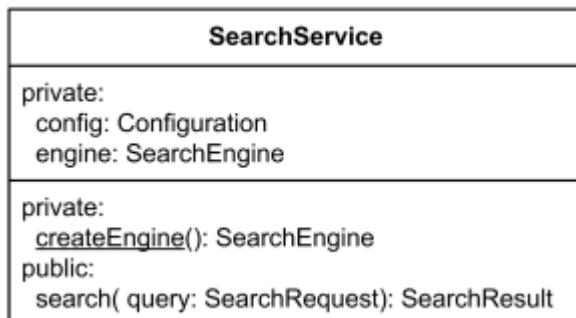
**Attributes** of a class are represented by instances of **Property** that are **owned by the class**. Some of these attributes may represent the **navigable ends of binary associations**.

**Objects** of a class must contain values for each attribute that is a member of that class, in accordance with the characteristics of the attribute, for example its type and multiplicity.



**Class** SearchService - implementation level details. The createEngine is **static** operation

Attributes or operations may be grouped by **visibility**. A visibility keyword or symbol in this case can be given once for multiple features with the same visibility.



**Class** SearchService - attributes and operations grouped by visibility

Additional compartments may be provided to show other details, such as **constraints**, or just to divide features.

---

## 5.4 RESPONSIBILITIES OF CLASSES

---

- **Knowing**
  - a. knowing about private encapsulated data
  - b. knowing about related objects
  - c. knowing about things it can derive or calculate
- **Doing**
  - a. ☐ doing something itself, such as creating an object or doing
- **Calculation**
  - a. initiating action in other objects
  - b. controlling and coordinating activities in other objects

### Examples

- a Sale is responsible for knowing its total (POS example)
- a Sale is responsible for creating SalesLineItems

---

## 5.5 ABSTRACT CLASS

---

**Abstract class** was defined as class that can't be directly instantiated. Abstract class exists only for other classes to inherit from and to support reuse of the features declared by it. No object may be a direct instance of an abstract class, although an object may be an indirect instance of one through a subclass that is nonabstract.

We can probably relate definition of abstract classifier to abstract class. We may assume that in UML 2.x **abstract class** does not have complete declaration and "typically" can not be instantiated. An abstract



class is intended to be used by other classifiers (e.g., as the target of generalization relationships).

Attempting to create an instance of an abstract class is **undefined** - some languages may make this action illegal, others may create a partial instance for testing purposes.

The name of an **abstract class** is shown in **italics**.



**Class** SearchRequest is **abstract class**

An abstract classifier can also be shown using the keyword **{abstract}** after or below the name.

---

## 5.6 STANDARD CLASS STEREOTYPES

---

Standard class stereotypes include:

- «focus»
- «auxiliary»
- «type»
- «utility»

### «focus»

**Focus** is class that defines the core logic or control flow for one or more supporting classes. The supporting classes may be defined either explicitly using auxiliary classes or implicitly by dependency relationships.

Focus classes are typically used for specifying the core business logic or control flow of components during design phase.

### «auxiliary»

**Auxiliary** is class that supports another more central or fundamental class, typically by implementing secondary logic or control flow. The class that the auxiliary supports may be defined either explicitly using a focus class or implicitly by a dependency relationship.

Auxiliary classes are typically used for specifying the secondary business logic or control flow of components during design phase.

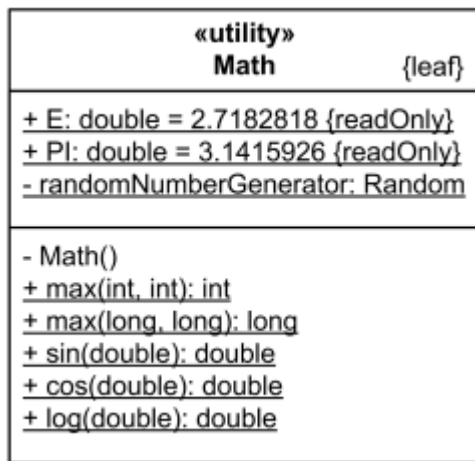
### «type»

**Type** is class that specifies a domain of objects together with the operations applicable to the objects, without defining the physical implementation of those objects.

Type may have attributes and associations. Behavioral specifications for type operations may be expressed using, for example, activity diagrams. An object may have at most one implementation class, however it may conform to multiple different types.

### «utility»

**Utility** is class that has only class scoped static attributes and operations. As such, utility class usually has no instances.



Math is **utility** class - having static attributes and operations (underlined)

### Visibility of attributes and operations

- **Public (+):** member of class is directly accessible to other objects; part of public interface
- **Private (-)** : member is hidden from public use; only accessible to other members within the class
- **Protected (#):** special visibility scope applicable to inheritance only

### Relational among classes

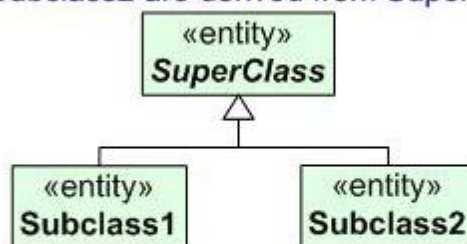
A class may be involved in one or more relationships with other classes. A relationship can be one of the following types:

#### Inheritance (or Generalization):

- Represents an “is-a” relationship.
- An abstract class name is shown in italics.
- *SubClass1* and *SubClass2* are specializations of *SuperClass*.

Generalization

Inheritance (Generalization): Subclass1 and Subclass2 are derived from SuperClass

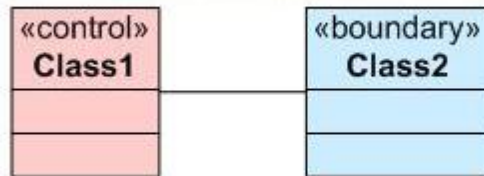


The relationship is displayed as a solid line with a hollow arrowhead that points from the child element to the parent element

### Association

- **Simple association:**
  - o A structural link between two peer classes.
  - o There is an association between *Class1* and *Class2*

## Simple Association between Class1 and Class2



The relationship is displayed as a solid line connecting two classes

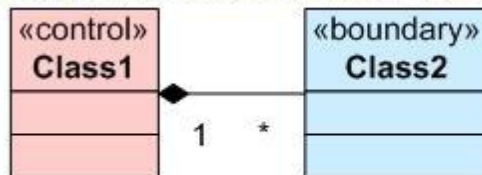
Associations can be one of the following two types or not specified.

**Composition:**

Represented by an association line with a solid diamond at the tail end. A composition models the notion of one object "owning" another and thus being responsible for the creation and destruction of another object. A special type of aggregation where parts are destroyed when the whole is destroyed.

- Objects of *Class2* live and die with *Class1*.
- *Class2* cannot stand by itself.

## Composition between Class1 and Class2



The relationship is displayed as a solid line with a **filled** diamond at the association end, which is connected to the class that represents the whole, or composite.

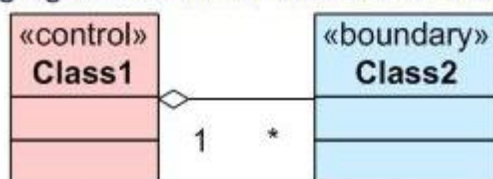
**Aggregation:**

Represented by an association line with a hollow diamond at the tail end. An aggregation models the notion that one object uses another object without "owning" it and thus is not responsible for its creation or destruction.

It is a special type of association. It represents a **“partof”** relationship.

- *Class2* is part of *Class1*.
- Many instances (denoted by the \*) of *Class2* can be associated with *Class1*.
- Objects of *Class1* and *Class2* have separate lifetimes.

## Aggregation between Class1 and Class2



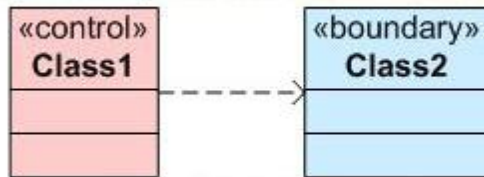
The relationship is displayed as a solid line with a **unfilled** diamond at the association end, which is connected to the class that represents the aggregate.

## 5.7 DEPENDENCY

A dotted line with an open arrowhead that shows one entity depends on the behavior of another entity. Typical usages are to represent that one class instantiates another or that it uses the other as an input parameter.

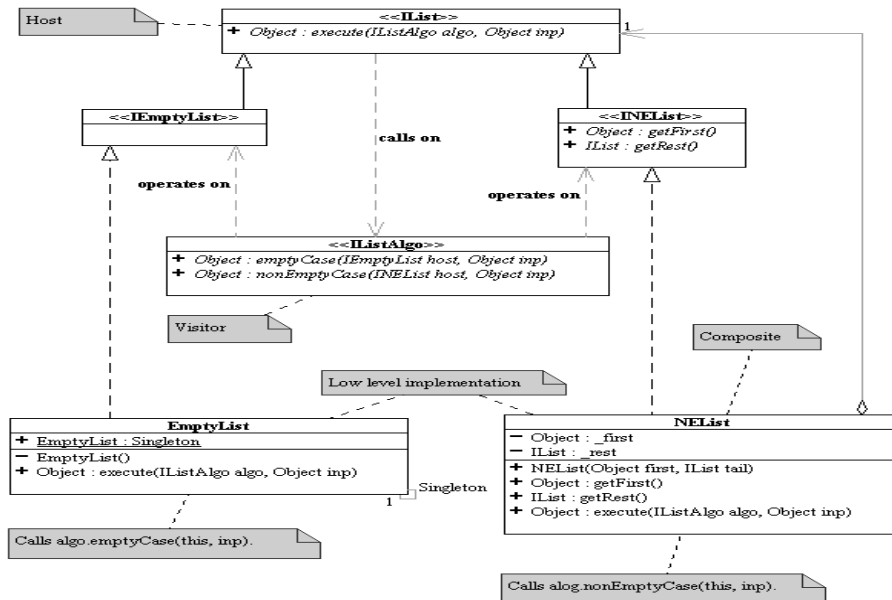
- It exists between two classes if changes to the definition of one may cause changes to the other (but not the other way around).
- *Class1* depends on *Class2*

Dependency between Class1 and Class2



The relationship is displayed as a dashed line with an open arrow.

Here is an example of a UML class diagram that holds most of the more common features. The above diagram contains classes, interfaces, inheritance and implementation lines, aggregation lines, dependency lines and notes



# ASSOCIATION CLASSES

## Unit Structure

- 6.1 Associations
- 6.2 Association Classes
- 6.3 Association Ends
- 6.4 Dependency

---

## 6.1 ASSOCIATIONS

---

As discussed in earlier. Association defines a type of link and is a general relationship between classes. For example, the project management system involves various general relationships, including manage, lead, execute, input, and output between projects, managers, teams, work products, requirements, and systems. Consider, for example, how a project manager leads a team.

### Binary associations

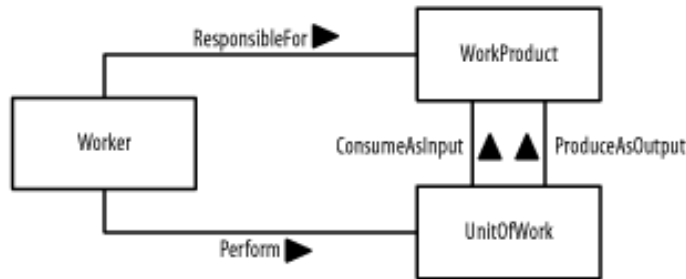
A binary association relates two classes. For example, one binary relationship in the project management system is between individual workers and their units of work, and another binary relationship is between individual workers and their work products.

In a UML class diagram, a binary association is shown as a solid-line path connecting the two related classes. A binary association may be labeled with a name. The name is usually read from left to right and top to bottom; otherwise, it may have a small black solid triangle next to it where the point of the triangle indicates the direction in which to read the name, but the arrow is purely descriptive, and the name of the association should be understood by the classes it relates.

Figure A shows various associations within the project management system using the most basic notation for binary associations. The associations in the figure are as follows:

- A worker is responsible for work products and performs units of work
- Units of work consume work products as input and produce work products as output.

Notice that a binary association should be named using a verb phrase.

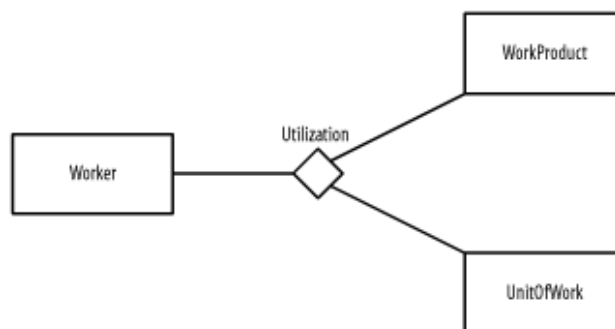
**Figure A. Binary associations**

### N-ary associations

An n-ary association relates three or more classes. For example, in the project management system, the use of a worker involves the worker, her units of work, and her associated work products.

In a UML class diagram, an n-ary association is shown as a large diamond with solid-line paths from the diamond to each class. An n-ary association may be labeled with a name. The name is read in the same manner as for binary associations, described in the previous section.

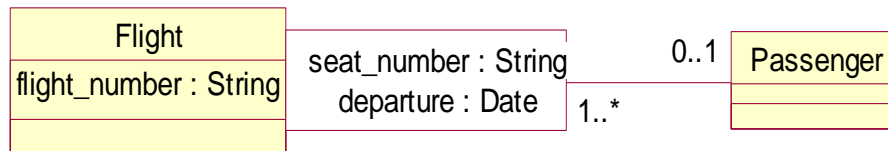
Figure B shows an n-ary association associated with the project management system using the most basic notation for n-ary associations. This association states that utilization involves workers, units of work, and work products. As with a binary association, an n-ary association is also commonly named using a verb phrase. However, this is not always the case — for example, the n-ary Utilization association shown in Figure B is described using a noun rather than a verb, because it is named from our perspective rather than the perspective of one of the classes. That is, from our perspective, we want to understand a worker's utilization relative to the other classes. From the worker's perspective, a worker is responsible for work products and performs units of work.

**Figure B. N-ary association**

### Qualified Association

A qualified association has an attribute compartment (a **qualifier**) on one end of a binary association (an association can be qualified on both ends but this is rare). The compartment contains one or more attributes that can serve as an index key for traversing the association from the **qualified class** via qualifier to the **target class** on the opposite association end

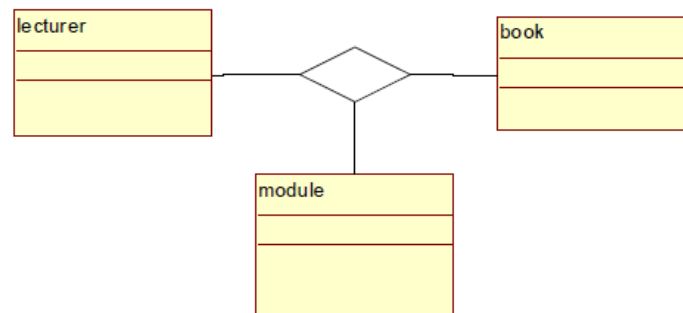
For e.g.



## Ternary Association

So far I have only mentioned associations which involve two classes, however you can have associations which have more than two classes involved called **n-ary** associations, specifically one involving three classes is called a **Ternary** association.

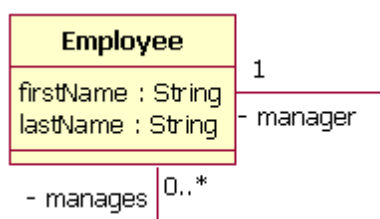
Let's consider an example, below is shown a ternary association between Lecturer, Book and Module. What does this mean? Well it indicates that there is a clear association between a specific lecturer, book and module. For example with this diagram we could be able to discover that Lecturer Joan uses the book 'UML and spirituality' on the module 'modern theology'. In other words we are defining a set of three specific values (L,B,M) in contrast to two specific values for an association involving two classes.



## Reflexive associations

We have now discussed all the association types. As you may have noticed, all our examples have shown a relationship between two different classes. However, a class can also be associated with itself, using a reflexive association. This may not make sense at first, but remember that classes are abstractions. Following figure shows how an Employee class could be related to itself through the manager/manages role. When a class is associated to itself, this does not mean that a class's instance is related to itself, but that an instance of the class is related to another instance of the class.

Example of a reflexive association relationship



The relationship drawn in Figure 14 means that an instance of Employee can be the manager of another Employee instance. However, because the relationship role of "manages" has a multiplicity of 0..\*; an Employee might not have any other Employees to manage.

## 6.2 ASSOCIATION CLASSES

Association classes may be applied to both binary and n-ary associations. Similar to how a class defines the characteristics of its objects, including their structural features and behavioral features, an association class may be used to define the characteristics of its links, including their structural features and behavioral features. These types of classes are used when you need to maintain information about the relationship itself.

In a UML class diagram, an association class is shown as a class attached by a dashed-line path to its association path in a binary association or to its association diamond in an n-ary association. The name of the association class must match the name of the association.

Figure C shows association classes for the binary associations in Figure A using the most basic notation for binary association classes. The association classes track the following information:

- The reason a worker is responsible for a work product
- The reason a worker performs a unit of work
- A description of how a unit of work consumes a work product
- A description of how a unit of work produces a work product.

**Figure C. Binary association classes**

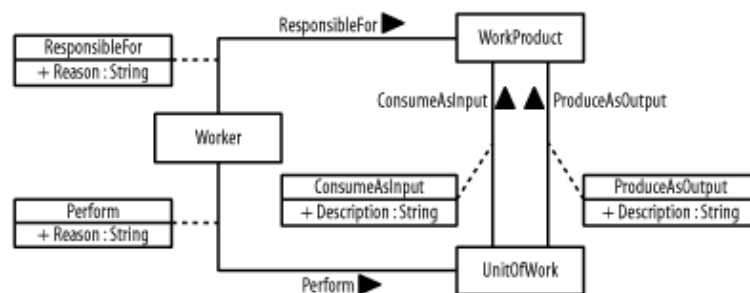
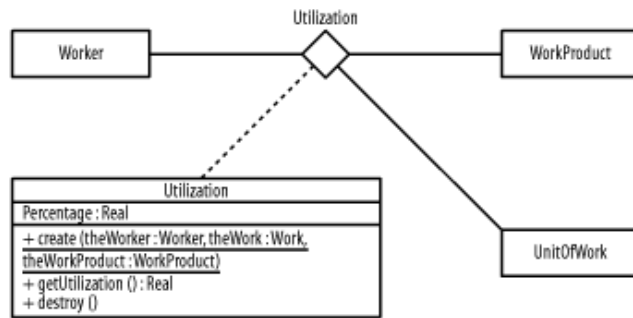


Figure D shows an association class for the n-ary association in Figure B using the most basic notation for n-ary association classes. The association class tracks a utilization percentage for workers, their units of work, and their associated work products.

**Figure D. N-ary association class**





## 6.3 ASSOCIATION ENDS

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

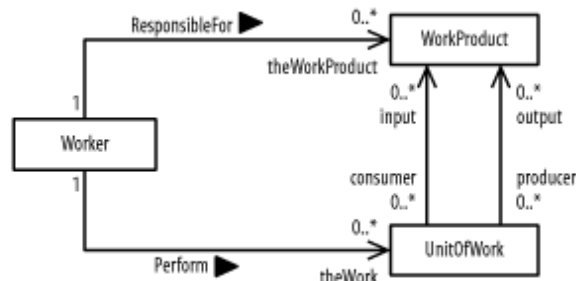
- Rolename
- Navigation arrow
- Multiplicity specification
- Aggregation or composition symbol
- Qualifier

### Rolenames

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

For example, a work product is seen as input by a unit of work where the unit of work is seen as a consumer by the work product; a work product is seen as output by a unit of work where the unit of work is seen as a producer by the work product, as shown in [Figure 3-13](#). I will continue to discuss this figure in the next sections. I particularly captured significant detail in one figure so that you can see how much the UMC enables you to communicate in a figure such as this.

**Figure E. Binary association ends**

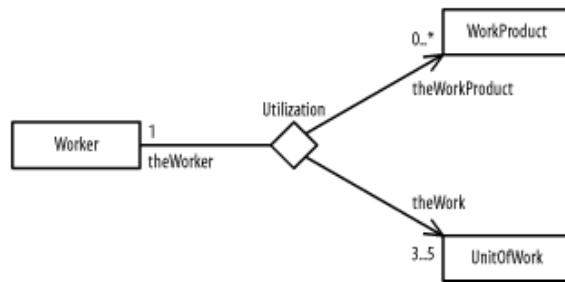


### Navigation

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

For example, given a worker, you can determine his work products and units of work. Thus, Figure E shows arrows pointing towards work product and units of work. Given a unit of work, you can determine its input and output work products; but given a work product, you are unable to identify which worker is responsible for it or which units of work reference it as input or output (as shown in Figure E by the lack of arrows pointing to the Worker class). Figure F shows navigation arrows applied to an n-ary association. Given a worker, you can reference his work products and units of work to determine his utilization, but given a work product or unit of work, you are unable to determine its utilization by a worker.

**Figure F . N-ary association ends**



## Multiplicity

Multiplicity (which is optional) indicates how many objects of a class may relate to the other classes in an association. Multiplicity is shown as a comma-separated sequence of the following:

- Integer intervals
- Literal integer values

Intervals are shown as a *lower-bound .. upper-bound* string in which a single asterisk indicates an unlimited range. No asterisks indicate a closed range. For example, 1 means one, 1..5 means one to five, 1, 4 means one or four, 0..\* and \* mean zero or more (or many), and 0..1 and 0, 1 mean zero or one. There is no default multiplicity for association ends. Multiplicity is simply undefined, unless you specify it. For example:

- A single worker is responsible for zero or more work products.
- A single work product is the responsibility of exactly one worker.
- A single worker performs zero or more units of work.
- A unit of work is performed by exactly one worker.
- A unit of work may input as a consumer zero or more work products and output as a producer zero or more work products.
- A work product may be consumed as input by zero or more units of work and produced as output by zero or more units of work.

All this is shown in Figure E. Continuing the example, utilization may be determined for a single worker who must have three to five units of work and zero or more work products, as shown in Figure 3-14.

To determine the multiplicity of a class, ask yourself how many objects may relate to a single object of the class. The answer determines the multiplicity on the other end of the association. For example, using figure E, if you have a single worker object, how many work products can a single worker object be responsible for? The answer is zero or more, and that is the multiplicity shown on the diagram

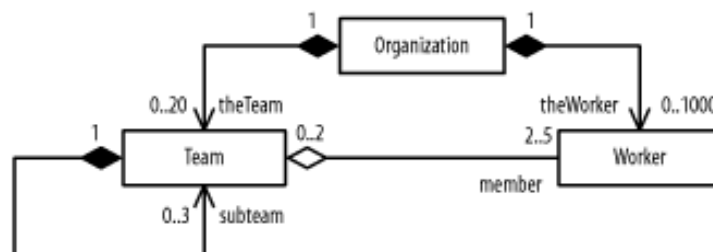
next to the WorkProduct class. Using Figure F, if you have a single worker, to how many work products can the single worker be related to determine the worker's utilization? The answer is zero or more, and that is the multiplicity shown on the diagram next to the WorkProduct class.

Another way to determine multiplicity is to ask how many objects of a class may relate to a single object of the class on the other end of an association, or to a single object of each class on the other ends of an n-ary association. The answer determines the multiplicity for the class. For example, using Figure E, how many work products is a single worker responsible for? The answer is zero or more; that is the multiplicity shown on the diagram next to the WorkProduct class. Also, using Figure F, to how many work products is a single worker and a single unit of work related to determine the worker's utilization? The answer is zero or more; that is the multiplicity shown on the diagram next to the WorkProduct class.

## Aggregation

Aggregation is whole-part relationship between an aggregate, the whole, and its parts. This relationship is often known as a has-a relationship, because the whole has its parts. For example, when you think of workers making up a team, you can say that a team has workers. Aggregation is shown using a hollow diamond attached to the class that represents the whole. This relationship that I've just mentioned between a team and its workers is shown in Figure G. Look for the hollow diamond to the right of the Team class. The filled-in diamonds represent composition, which I'll discuss next. As a UML rule, aggregation is used only with binary associations.

**Figure G. Aggregation and composition for associations**



Notice in Figure G that I've done something different with Team. I've created a circular relationship to allow for subteams. Such a circular relationship is known as a reflexive relationship, because it relates two objects of the same class.

## Composition

Composition, also known as composite aggregation, is a whole-part relationship between a composite (the whole) and its parts, in which the parts must belong only to one whole and the whole is responsible for creating and destroying its parts when it is created or destroyed. This relationship is often known as a contains-a relationship, because the whole contains its parts. For example, an organization contains teams and workers, and if the organization ceases to exist, its teams and workers also cease to exist. The specific individuals who represent the workers would still exist, but they would no longer be workers of the organization, because the organization would no longer exist. Composition is shown using a filled diamond attached to the class that represents the whole. The relationships between a team, its workers,

and an organization are shown in Figure G. The filled-in diamond at the endpoint of the subteam relationship in Figure G indicates that teams contain their subteams. As a UML rule, composition is used only with binary associations.

Notice how much information is being communicated in Figure G. It shows that an organization may contain 0 to 20 teams and 0 to 1,000 workers. Furthermore, each team has 2 to 5 workers and each worker may be a member of 0 to 2 teams. In addition, a team may contain 0 to 3 subteams.

To determine if you should use an aggregation or composition, ask yourself a few questions. First, if the classes are related to one another, use an association. Next, if one class is part of the other class, which is the whole, use aggregation; otherwise, use an association. For example, Figure G shows that workers are part of a team and organization, teams are part of an organization, and subteams are part of teams. Finally, if the part must belong to one whole only, and the whole is responsible for creating and destroying its parts, use composition; otherwise, simply use aggregation.

For example, Figure G shows that a team and worker must belong to one organization only, and the organization is responsible for determining (or creating and destroying) its teams and workers. It also shows that a subteam must belong to one team only, and the team is responsible for determining (or creating and destroying) its subteams. If this is unclear, keep things simple and use associations without aggregation or composition.

Composition also may be shown by graphically nesting classes, in which a nested class's multiplicity is shown in its upper-right corner and its rolename is indicated in front of its class name. Separate the rolename from the class name using a colon. Figure H uses the graphical nesting of teams and workers in organizations to communicate the same information as shown in Figure G.

**Figure H. Alternate form of composition for associations**

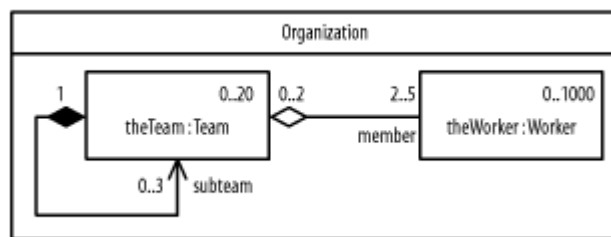
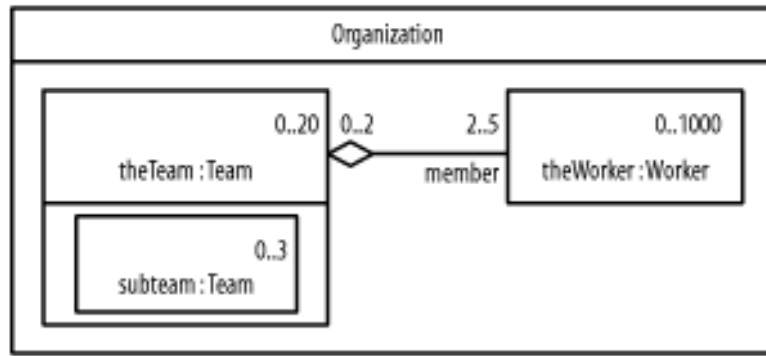


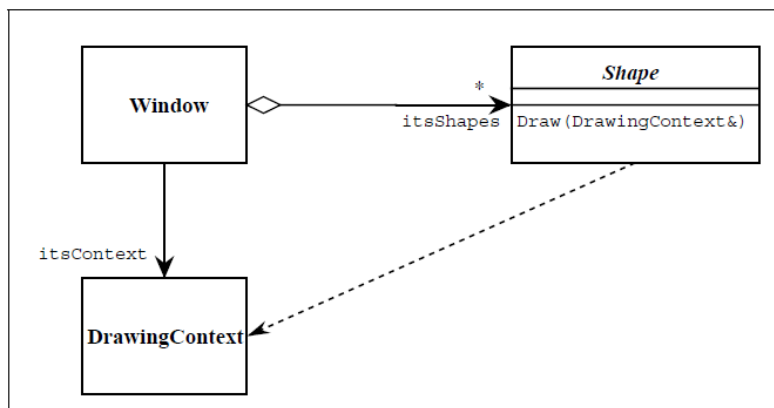
Figure I uses the graphical nesting of subteams within teams to communicate the same information as Figure I and Figure H. The result is a nested class inside a nested class.

**Figure I. Doubly nested composition for associations**



## 6.4 DEPENDENCY

Sometimes the relationship between a two classes is very weak. They are not implemented with member variables at all. Rather they might be implemented as member function arguments. Consider, for example, the Draw function of the Shape class. Suppose that this function takes an argument of type DrawingContext.

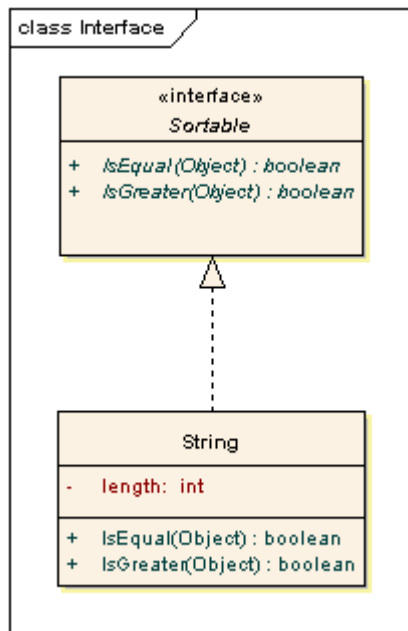


Above figure shows a dashed arrow between the Shape class and the DrawingContext class. This is the dependency relationship. This relationship simply means that Shape somehow depends upon DrawingContext. In C++ this almost always results in a #include.

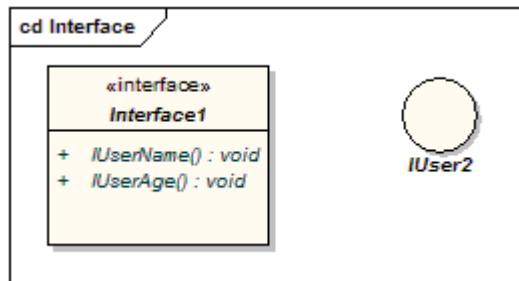
- Adornments on association: association names, association classes, qualified association, n-ary associations, ternary and reflexive association
- Dependency relationships among classes, notations
- Notes in class diagram, extension mechanisms, metadata, refinements, derived, data, constraint, stereotypes, package & interface notation.
- Object diagram notations and modeling, relations among objects (links)

### Interfaces

An interface is a specification of behavior that implementers agree to meet; it is a contract. By realizing an interface, classes are guaranteed to support a required behavior, which allows the system to treat non-related elements in the same way – that is, through the common interface.



Interfaces may be drawn in a similar style to a class, with operations specified, as shown below. They may also be drawn as a circle with no explicit operations detailed. When drawn as a circle, realization links to the circle form of notation are drawn without target arrows.



# STATE DIAGRAM

## Unit Structure

- 7.1 Introduction
- 7.2 States
- 7.3 Events
- 7.4 Composite States
- 7.5 Transitions

---

## 7.1 INTRODUCTION

---

This chapter focuses on state diagrams, also known as state chart diagrams, which depict the lifecycle of elements that make up a system. First, we introduce state diagrams and how they are used. Next, we go over states and their details. Finally, we discuss transitions between states and their details. Many details of state diagrams that were not fleshed out in previously are more fully elaborated here, and throughout the chapter, we include suggestions relating to state diagrams.

State modeling is a specialized type of behavioral modeling concerned with modeling the lifecycle of an element. You usually apply state modeling in conjunction with interaction and collaboration modeling to explore the lifecycle of interacting and collaborating elements.

---

## 7.2 STATES

---

As discussed in earlier, as elements communicate with one another within a society of objects, each element has a lifecycle in which it is created, knows something, can do something, can communicate with other elements to request processing of those other elements, can have other elements communicate with it to request processing of it, and is destroyed. A state is a specific condition or situation of an element during its lifecycle. Define the states for your elements. The current state of an element is called its active state, and the element is said to be "in" that state. There are various types of states, including simple, initial, and final states. The next few sections discuss these different types of states.

### Simple States

A simple state indicates a condition or situation of an element. For example, the project management system may be in one of the following simple states:

**Inactive:** Indicates that the project management system is not available to its users, because it is not started or has been shut down.

**Active:** Indicates that the project management system has been started and is available to its users.

**Suspended:** Indicates that the project management system has encountered some severe error, perhaps because it is running low on secondary storage and requires user intervention before becoming active again

**Notation:**

In the UML, a simple state is shown as a rectangle with rounded corners and labeled with the name of the state or a description of the situation of the element. Following figure shows the various states associated with the project management system.

**Simple states**



**Initial and Final States**

An initial state indicates the state of an element when it is created. In the UML, an initial state is shown using a small solid filled circle. A final state indicates the state of an element when it is destroyed. In the UML, a final state is shown using a circle surrounding a small solid filled circle (a bull's eye). Above figure updates with an initial state and final state. A state diagram may have only one initial state, but may have any number of final states

**Simple, initial, and final states**




---

## 7.3 EVENTS

---

An event is an occurrence, including the reception of a request. There are a number of different types of events within the UML.

- **CallEvent.** Associated with an operation of a class, this event is caused by a call to the given operation. The expected effect is that the steps of the operation will be executed.
- **SignalEvent.** Associated with a signal, this event is caused by the signal being raised.
- **TimeEvent.** An event caused by expiration of a timing deadline.



- **ChangeEvent.** An event caused by a particular expression (of attributes and associations) becoming true.

An event is represented by its name.

For example, the project management system may respond to the following events:

- **Startup:** Indicates that the project management system will become active if it is inactive.
- **Shutdown:** Indicates that the project management system will become inactive if it is active.
- **Severe Error:** Indicates that the project management system has encountered a severe error, perhaps because it is running low on secondary storage, and will become suspended if it is active.
- **Reset:** Indicates that the project management system will become active if it is suspended.

In the UML, an event is described using the following UML syntax:

*event\_name (parameter\_list) [guard]*

in which:

- **event\_name:** Is the name of the event. An event usually has the same name as an operation of the element to which the state diagram pertains; therefore when the element receives the event, that operation is invoked.
- **parameter\_list:** Is optional, and is a comma-separated list that indicates the parameters passed to the event. Each parameter may be an explicit value or a variable. The parentheses are not used when the event does not require any parameters.
- **guard:** Is optional, and indicates a condition that must be satisfied for the transition to fire, or occur. The square brackets are not shown when a guard is not specified.

The UML also allows you to show an event using pseudocode or another language. For example, you can use the syntax of Java, C++, C#, or some other programming language.

Following is an example of an event defined using the syntax just shown. The event informs the project management system to start up:

### **Startup**

If this event requires the user's identification, you can update the event to the following:

### **Startup (UserID)**

If the project management system responds to this event only if it can start up, perhaps if enough memory is available, you can update the transition to the following:

Startup (UserID) [Enough memory is available]

## 7.4 COMPOSITE STATES

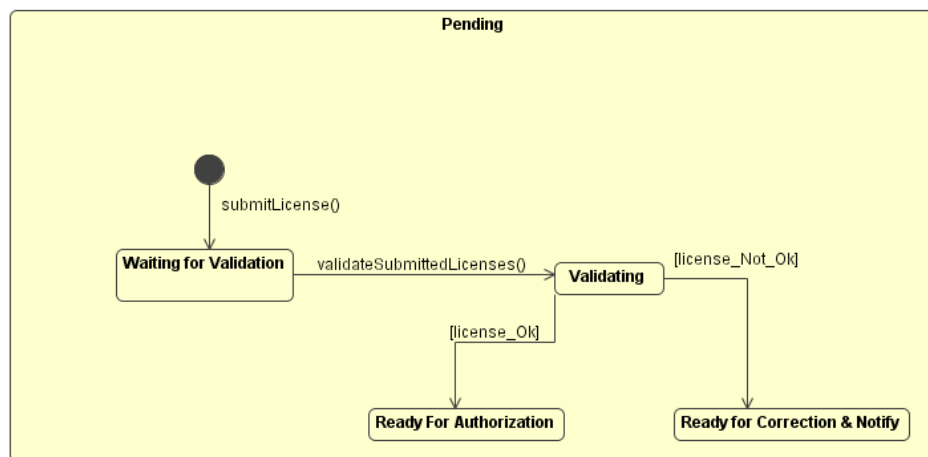
A composite state is simply a state that contains one or more statechart diagrams.

Composite states may contain either:

- 1) a set of mutually exclusive states: literally like embedding a statechart diagram inside a state
- 2) a set of concurrent states: different states divided into regions, active at the same time

A composite state is also called a super-state, a generalized state that contains a set of specialized states called substates.

A composite state (super-state) may be decomposed into two or more lower-level states (sub-states). All the rules and notation are the same for the contained substates as for any statechart diagram. Decomposition may have as many levels as needed.

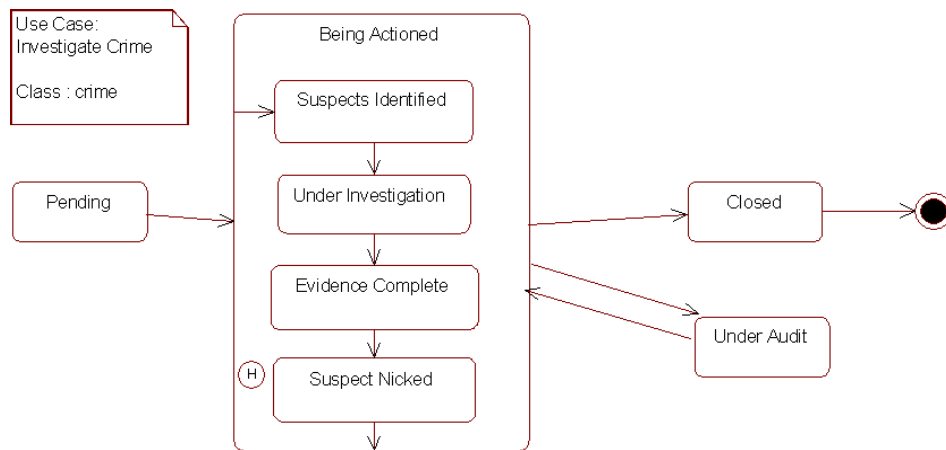


**History States:** A flow may require that the object go into a trance, or wait state, and on the occurrence of a certain event, go back to the state it was in when it went into a wait state—its last active state. This is shown in a State diagram with the help of a letter H enclosed within a circle.



Take the following example.

A criminal investigation starts in the "pending" state. Once it switches to the "Being Actioned" state, it can be in a number of substates. However, at random intervals, the case can be audited. During an audit, the investigation is briefly suspended. Once the audit is complete, the investigation must resume at the state from where it was before the audit. The simple "history notation" (a "H" in a circle) allows us to do this, as follows:



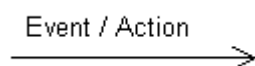
## 7.5 TRANSITIONS

Given the states of an element, how are they related to one another? Transitions address this question. As discussed earlier, transitions between states occur as follows:

1. An element is in a source state.
2. An event occurs.
3. An action is performed.
4. The element enters a target state.

When an event occurs, the transition is said to fire. In the UML, a transition is shown as a solid line from a source state to a target state labeled with the event followed by a forward slash followed by the action, where the event and action are optional and the forward slash is used only if an action is shown.

Simply it is an arrow indicating the Object to transition from one state to the other. The actual trigger event and action causing the transition are written beside the arrow, separated by a slash. Transitions that occur because the state completed an activity are called "**triggerless**" transitions. If an event has to occur after the completion of some event or action, the event or action is called the guard condition. The transition takes place after the guard condition occurs. This guard condition/event/action is depicted by square brackets around the description of the event/action (in other words, in the form of a Boolean expression).



### Entry Actions

More than one event can trigger a transition of an object into the same state. When the same action takes place in all events that goes into a state, the action may be written once as entry action.

Notation:

- 1) Use the keyword entry followed by a slash and the actions to be performed every time the state is entered
- 2) Entry actions are part of internal transitions compartment

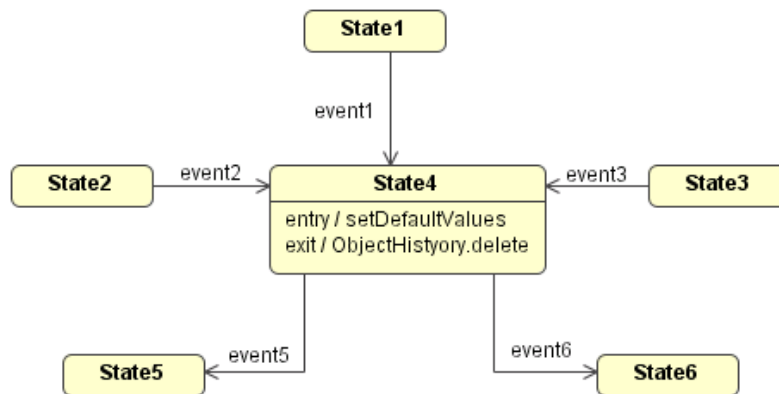
## Exit Actions

The same simplification can be used for actions associated with events that trigger a transition out of a state. They are called exit actions.

Notation:

- 1) Use the keyword **exit** followed by a slash and the actions performed every time the state is exited
- 2) Exit actions are part of the internal transitions compartment Only when the action takes places every time the state is exited.

For example.



## Send Events

Sometimes the object modelled by the statechart needs to send a message to another object, in this case the outgoing event must define which is the receiving object. Also, this event must be caught by the receiving object. A message send to another object is called a send event.

Notation: provide the object name before the action expression with a period separating both



# SEQUENCE DIAGRAM

## Unit Structure

- 8.1 Introduction
- 8.2 Notations
- 8.3 Communication
- 8.4 Types of Messages
- 8.5 Conditional Messaging

---

## 8.1 INTRODUCTION

---

A sequence diagram shows elements as they interact over time, showing an interaction or interaction instance. Sequence diagrams are organized along two axes: the horizontal axis shows the elements that are involved in the interaction, and the vertical axis represents time proceeding down the page. The elements on the horizontal axis may appear in any order.

---

## 8.2 NOTATIONS

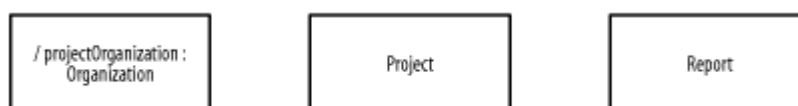
---

Sequence diagrams are made up of a number of elements, including class roles, specific objects, lifelines, and activations. All of these are described in the following subsections.

### Class roles

In a sequence diagram, a class role is shown using the notation for a class as defined in previously, but the class name is preceded by a forward slash followed by the name of the role that objects must conform to in order to participate within the role, followed by a colon. Other classes may also be shown as necessary, using the notation for classes. Class roles and other classes are used for specification-level collaborations communicated using sequence diagrams. Following figure shows the projectOrganization class role as well as the Project and Report classes.

**Figure A. Class role and two classes**

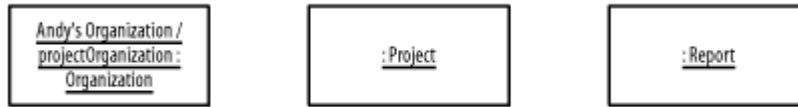


### Specific objects

In a sequence diagram, a specific object of a class conforming to a class role is shown using the notation for objects, but the object name is followed by a forward slash followed by the name of the role followed by a colon followed by the class name, all fully underlined. Other objects may also be shown as necessary using the notation for objects.

Specific objects conforming to class roles and other objects are used for instance-level collaborations communicated using sequence diagrams. Figure B shows that Andy's organization plays the role of an organization that contains a project that is the subject of the report. Figure B also shows anonymous Project and Report objects.

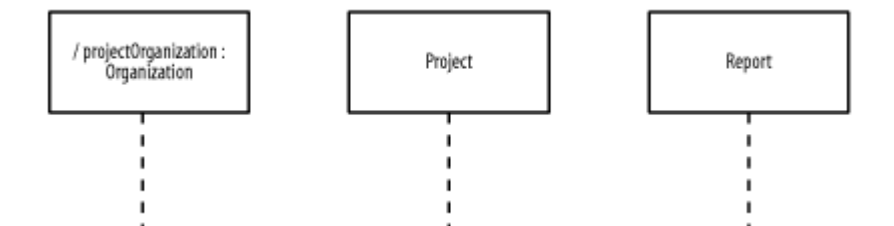
**Figure B. Object conforming to a class role**



## Lifelines

A lifeline, shown as a vertical dashed line from an element, represents the existence of the element over time. Figure C shows lifelines for the class role (projectOrganization) and classes (Project and Report) in Figure A. Lifelines may also be shown for the objects in Figure B.

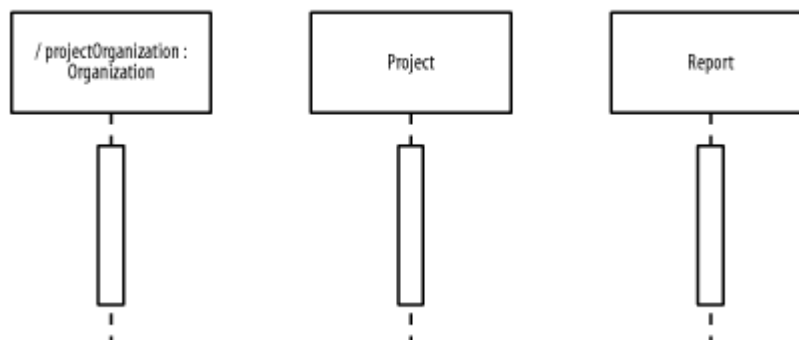
**Figure C. Lifelines**



## Activations

An optional activation, shown as a tall, thin rectangle on a lifeline, represents the period during which an element is performing an operation. The top of the rectangle is aligned with the initiation time, and the bottom is aligned with the completion time. Figure D shows activations for the class roles in Figure A, where all the elements are simultaneously performing operations. Activations may also be shown for the objects in Figure B.

**Figure D. Activations**



---

## 8.3 COMMUNICATION

---

In a sequence diagram, a communication, message, or stimulus is shown as a horizontal solid arrow from the lifeline or activation of a sender to the lifeline or activation of a receiver. In the UML, communication is described using the following UML syntax:

```
[guard]    *[iteration]    sequence_number    :    return_variable    :=
operation_name
```

```
(argument_list)
```

in which:

**guard**

Is optional and indicates a condition that must be satisfied for the communication to be sent or occur. The square brackets are removed when no guard is specified.

**iteration**

Is optional and indicates the number of times the communication is sent or occurs. The asterisk and square brackets are removed when no iteration is specified.

**sequence\_number**

Is an optional integer that indicates the order of the communication. The succeeding colon is removed when no sequence number is specified. Because the vertical axis represents time proceeding down the page on a sequence diagram, a sequence number is optional.

**return\_variable**

Is optional and indicates a name for the value returned by the operation. If you choose not to show a return variable, or the operation does not return a value, you should also omit the succeeding colon and equal sign.

**operation\_name**

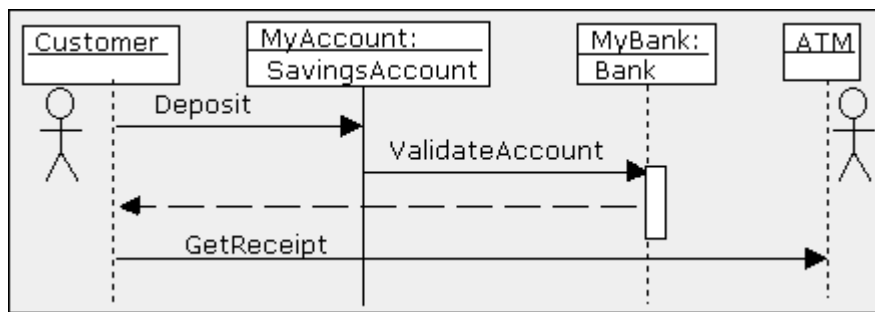
Is the name of the operation to be invoked.

**argument\_list**

Is optional and is a comma-separated list that indicates the arguments passed to the operation. Each parameter may be an explicit value or a return variable from a preceding communication. If an operation does not require any arguments, the parentheses are left empty.

### Example

In some ways, a sequence diagram is like a stack trace of object messages. Below is a sample sequence diagram



The Sequence diagram allows the person reading the documentation to **follow the flow of messages** from each object. The vertical lines with the boxes on top represent instances of the classes (or objects).

The label to the left of the colon is the instance and the label to the right of the colon is the class. The horizontal arrows are the messages passed between the instances and are read from top to bottom.

Here

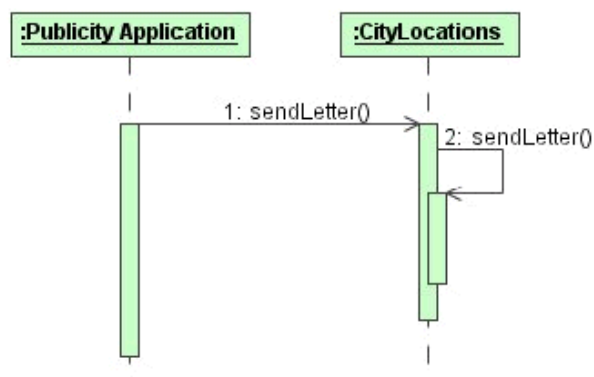
- a customer (user) depositing money into MyAccount which is an instance of Class SavingsAccount.
- Then MyAccount object Validates the account by asking the Bank object, MyBank to ValidateAccount.
- Finally, the Customer Asks the ATM object for a Receipt by calling the ATM's operation GetReceipt.

The white rectangle indicate the scope of a method or set of methods occurring on the Object My Bank. The dotted line is a return from the method ValidateAccount.

### Recursion (Repetition)

An object might also need to call a message recursively, this means to call the same message from within the message.

- 1) Suppose that cityLocations is defined in the class diagram as a set of one or more apartments or houses
- 2) A letter could be sent to all apartments in a location as shown



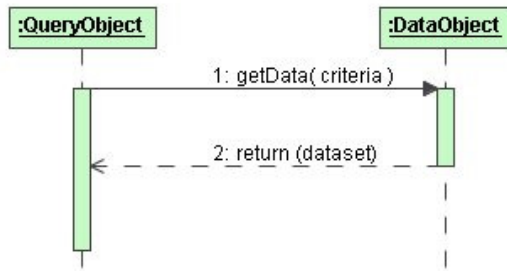
## 8.4 TYPES OF MESSAGES

### • Synchronous Messages

- 1) assumes that a return is needed
- 2) sender waits for the return before proceeding with any other activity



- 3) represented as a full arrow head
- 4) return messages are dashed arrows



### • Asynchronous Messages

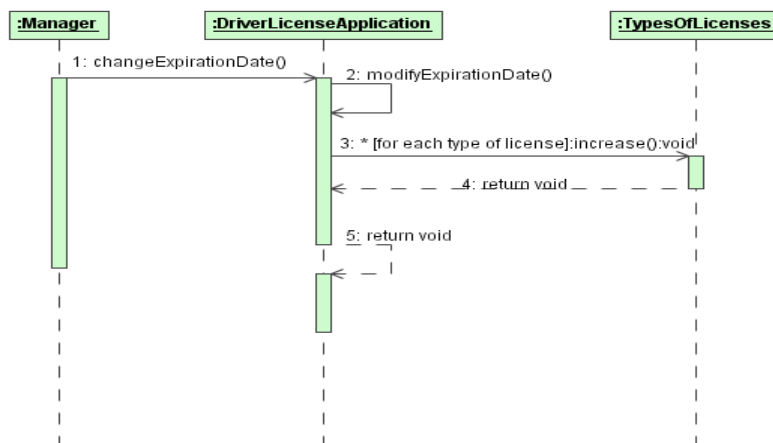
- 1) does not wait for a return message
- 2) exemplified by signals
- 3) sender only responsible for getting the message to the receiver
- 4) usually modeled using a solid line and a half arrowhead to distinguish it from the full arrowhead of the synchronous message



### • Self-Reference Message

A self-reference message is a message where the sender and receiver are one and the same object.

- 1) in a self-reference message the object refers to itself when it makes the call
- 2) message 2 is only the invocation of some procedure that should be executed



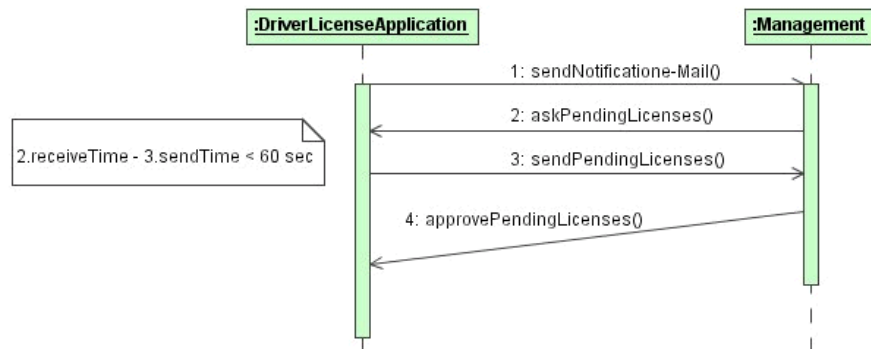
### • Timed Messages

- 1) messages may have user-defined time attributes, such as

sentTime or receivedTime

- 2) user-defined time attributes must be associated with message numbers
  - 3) instantaneous messages are modeled with horizontal arrows
  - 4) messages requiring a significant amount of time, it is possible to slant the arrow from the tail down to the head
- Object Creation and Destruction

For Example



For messages 1, 2 and 3 the time required for their execution is considered equal to zero.

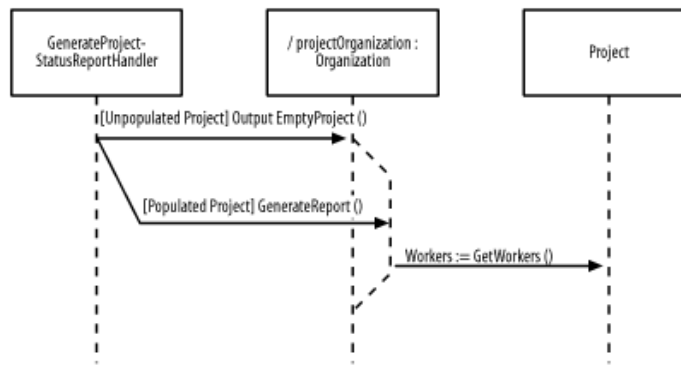
Message 4 requires more time (time > 0) for its execution.

## 8.5 CONDITIONAL MESSAGING

In a sequence diagram, conditionality (which involves communicating one set of messages or stimuli rather than another set of messages or stimuli) within a generic-form interaction is shown as multiple communications leaving a single point on a lifeline or activation, with the communications having mutually exclusive guard expressions. A lifeline may also split into two or more lifelines to show how a single element would handle multiple incoming communications, and the lifelines would subsequently merge together again.

Following figure shows the Generate Project-Status Report interaction and collaboration description where the GenerateProject-StatusReportHandler class requests that the projectOrganization class role indicate that the project is empty if the project is a newly created or unpopulated project, and the GenerateProject-StatusReportHandler class requests that the projectOrganization class role continue generating information for the report element if the project is not a newly created or populated project. In this figure, only the first communication is shown for actually generating the report. If there are no other communications for actually generating the report, the GenerateReport communication may go to the same lifeline as the OutputEmptyProject communication. we use different lifelines in the figure because each lifeline represents a different path of execution.

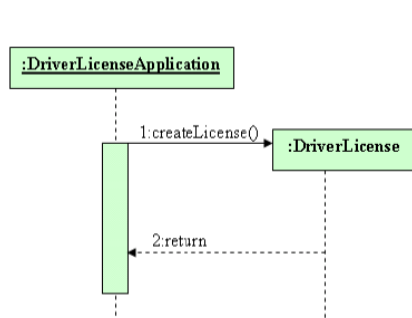
## Sequence diagram conditionality within a generic-form interaction



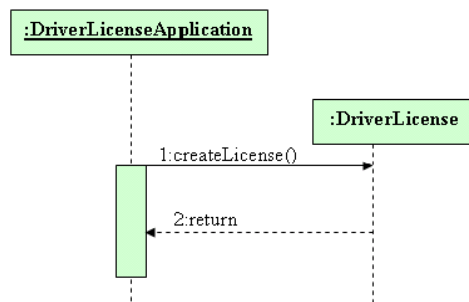
**Object Creation:** if the object is created during the sequence execution it should appear somewhere below the top of the diagram.

For example.

Alternative A

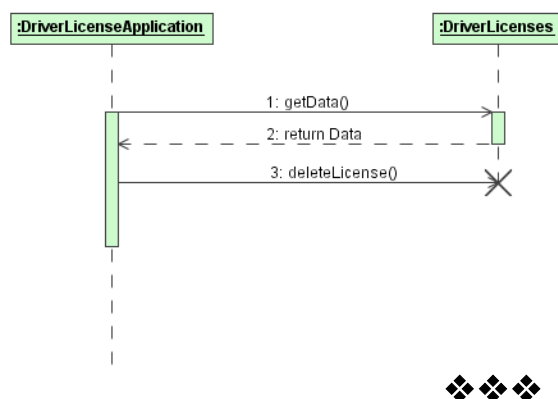


Alternative B



**Object Destruction:** if the object is deleted during the sequence execution, place an X at the point in the object lifeline when the termination occurs.

For example.



# COLLABORATION DIAGRAM

## Unit Structure

- 9.1 Introduction
- 9.2 Definition
- 9.3 Notations
- 9.4 Repetition
- 9.5 Conditional Messaging
- 9.6 Object Creation and Destruction

---

## 9.1 INTRODUCTION

---

A collaboration diagram shows elements as they interact over time and how they are related. That is, it shows a collaboration or collaboration instance. While sequence diagrams are time-oriented and emphasize the overall flow of an interaction, collaboration diagrams are time- and space-oriented and emphasize the overall interaction, the elements involved, and their relationships. Sequence diagrams are especially useful for complex interactions, because you read them from top to bottom. Collaboration diagrams are especially useful for visualizing the impact of an interaction on the various elements, because you can place an element on a diagram and immediately see all the other elements with which it interacts.

---

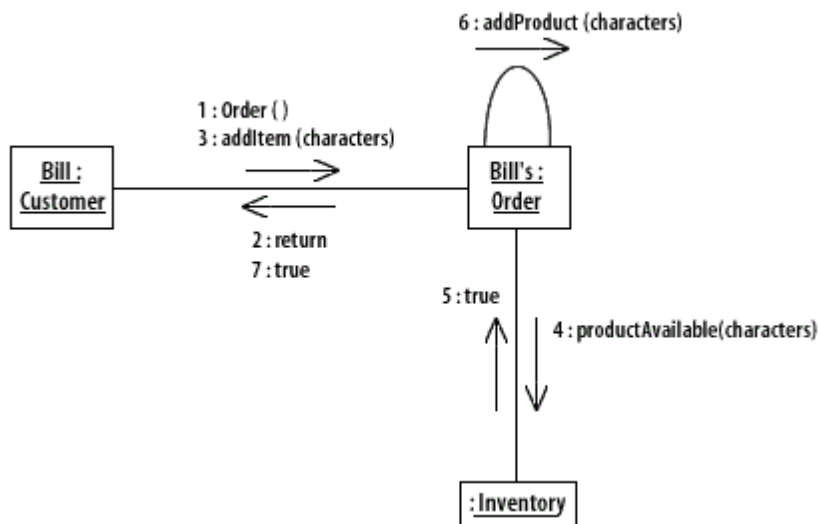
## 9.2 DEFINITION

---

A collaboration diagram shows the interactions organized around the structure of a model, using either:

- a) Classifiers (e.g. classes) and associations, or
- b) Instances (e.g. objects) and links.
  - 1) is an interaction diagram
  - 2) is similar to the sequence diagram
  - 3) reveals both structural and dynamic aspects of a collaboration
  - 4) reveals the need for the associations in the class diagram

For example.



### 9.3 NOTATIONS

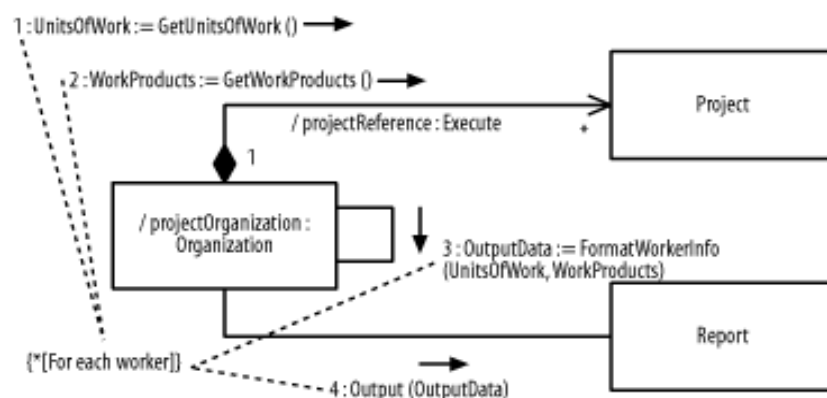
- 1) A collaboration diagram shows a graph of either instances linked to each other or classifiers and associations.
- 2) Navigability is shown using arrow heads on the lines representing links.
- 3) An arrow next to a line indicates a stimuli or message flowing in the given direction.
- 4) The order of interaction is given with a number

### 9.4 REPETITION

In a collaboration diagram, repetition (which involves repeating a set of messages or stimuli) within a generic-form interaction is shown as a property. An iteration expression indicating the number of times the communications occur may be enclosed in a pair of braces (`{}`) and attached to the communications to which it applies using dashed lines.

For example.

Following figure shows the Generate Project-Status Report interaction and collaboration description. The figure uses an iteration expression that causes the `GenerateProject-StatusReportHandler` class to retrieve each worker's units of work and list of work products, to format this information, and to output the formatted information to the report element.



## 9.5 CONDITIONAL MESSAGING

In a collaboration diagram, conditional messaging which involves communicating one set of messages or stimuli rather than another set of messages or stimuli — within a generic-form interaction is shown using the dot notation where the communication at a specific level indicates the guard expression that must be satisfied for the next level of communications to occur. For example, between elements A and B, communication 1 may be labeled with a guard expression and is followed by communication 2. Communication 1 from A to B may trigger communications 1.1 and 1.2 between B and C. Because of the guard condition on communication 1, the following two scenarios are possible:

### The guard condition is satisfied

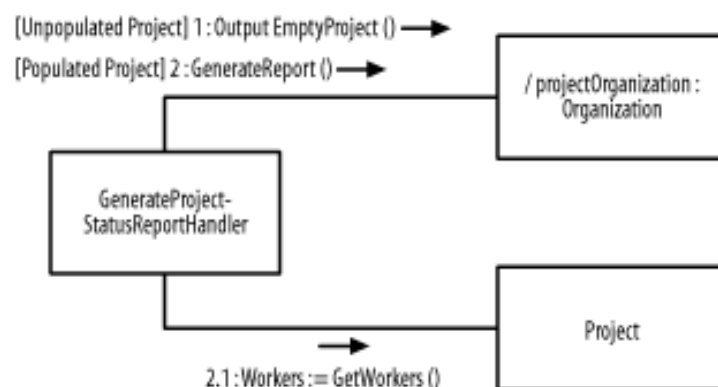
Communication 1 occurs between A and B, triggering 1.1 and 1.2 between B and C. The overall sequence of communications is then 1, 1.1, 1.2, followed by 2. Communication 2 comes last, because everything associated with communication 1 must precede it.

### The guard condition is not satisfied

Communication 1 does not occur, and thus neither do communications 1.1 and 1.2. However, communication 2, does occur because it is not protected by the same guard condition as communication 1.

Following figure shows the Generate Project-Status Report interaction and collaboration description where the GenerateProject-StatusReportHandler class requests that the projectOrganization class role indicate that the project is empty if the project is newly created or unpopulated; and the GenerateProject-StatusReportHandler class requests that the projectOrganization class role continue generating information for the report element if the project is not a newly created or populated project. In this figure, only the first communication is shown for actually generating the report.

### Collaboration diagram conditionality within a generic-form interaction



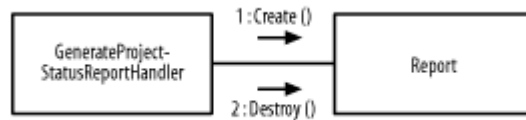
---

## 9.6 OBJECT CREATION AND DESTRUCTION

---

A communication that creates an element and a communication that destroys an element are simply shown like any other communication.

For example.



# 10

## DEVELOPING DYNAMIC SYSTEMS

### Unit Structure

- 10.1 Introduction
- 10.2 Diagrammatic Views support by UML
- 10.3 From Inception to Transition
- 10.4 Software Architecting
- 10.5 Architectural Views in UML

---

### 10.1 INTRODUCTION

---

In the remainder of this work we will primarily use the Unified Modeling Language (UML) as the foundation of our development model. In this section we will briefly describe UML, which is currently the leading object-oriented analysis and design model. UML's views are for the most part graphical diagrams. However, it must be understood that UML is only **a part of** our development model since it is missing important concepts.

UML [Booch-Jacobson-Rumbaugh 1998] is the result of a collaboration of numerous companies and OO modeling experts, led by Rational Software Corporation, and it borrows heavily from Booch [Booch 1994], OMT [Rumbaugh et al. 1991], and other OO models [Coad-Yourdon 1991a][Coad-Yourdon 1991b][Jacobson 1992]. The lead designers of UML were Booch, Jacobson and Rumbaugh. "UML is a language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems" [Booch-Rumbaugh- Jacobson 1997]. These different but overlapping uses of the model (or language) can only be achieved by supporting a variety of views. Some of these views (representations) can be seen in Figure. This figure shows four of the most popular graphical design techniques which are currently supported

by UML; the collaboration diagram, class diagram, use-case diagram, and state diagram.

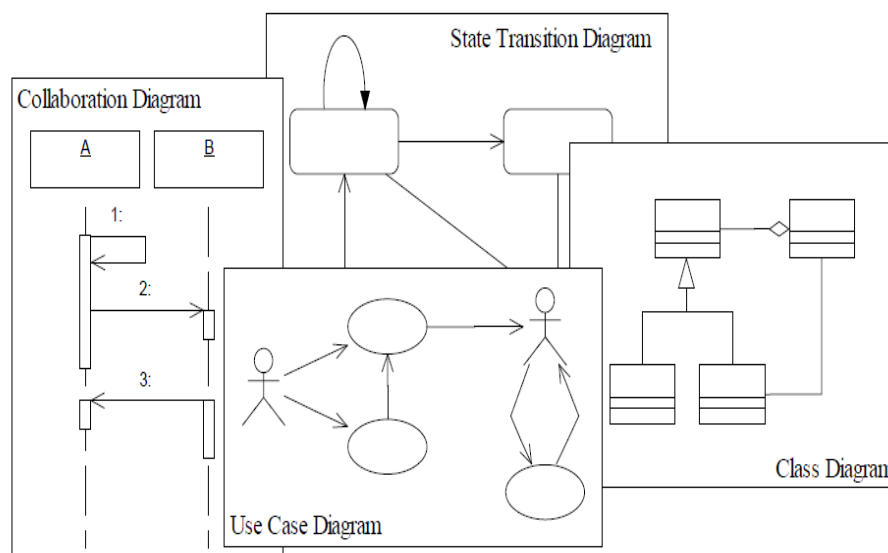
These views and others are briefly explained in the following. A more detailed description of some of these views is given later. Please refer to [Booch-Rumbaugh-Jacobson 1997] for a detailed description of UML (Version 1.2).

**Use Case** (e.g. Use Case Diagrams): Depicts the interaction between the user and the system or between one system and another. In doing so, use cases provide a high-level view of the usage of a system which frequently shows the interaction of multiple functions of the system.

E.g. the task of editing a document involves the functions *open document*, *edit document*, and *save document*.

**Interaction** (e.g. Sequence and Collaboration Diagrams): Sometimes also referred to as Mini- Uses. Interaction diagrams show concrete examples (e.g. test case) of how components communicate. They can often be seen as an actual test cases which involve the use of a single function (see use case). A function can refer to the GUI (e.g. open file) or to subcomponents of the system.

**Objects and Classes** (e.g. Class Diagrams, CRC Cards): Classes are the most central view in UML (and all other OO models). They depict the relationships between classes and objects which are the smallest stand-alone components in OO. Relationships can depict instances, part-of relationships, dependencies, and others.




---

## 10.2 DIAGRAMMATIC VIEWS SUPPORT BY UML

---

**Packages** (e.g. Package Diagram): Packages are used to group classes into layers and partitions. As such they show the functional decomposition of a system.

**State Transition** (e.g. State and Activity Diagrams): This well-known technique is used in UML to describe the states a class can go through.



In UML, state diagrams are restricted to a single classes only. Activity Diagrams are a generalization of state diagrams in that they can also be used to depict events or other ‘transitional’ elements.

**Deployment** (e.g. Deployment Diagrams): Shows the relationship of the components of the system during deployment. As such it presents a physical view of the system. It is, therefore, frequently used to depict the component dependency of the actual implementation.

UML does, however, not cover the entire set of useful stakeholder views. In addition to the views described above, the following are often considered to be very important extensions:

**Information Flow** (e.g. Data Flow Diagrams): Shows the functional flow of information. This view is particularly useful to users and customers since the human mind tends to think in terms of flow of information (documents, etc.). However, the decomposition yielded through this type of view is not well matched to object-oriented design.

**Interface/Dialog** (e.g. Interface Flow Diagrams): Describes the usage of the user interface of the system which again does not reflect any OO structure. Instead, it reflects the use cases in describing which functionality needs to be available where in the user interface (GUI). The Interface/Dialog Diagram reflects what the user sees of the system.

An object constraint language (OCL) [Booch-Rumbaugh-Jacobson 1997] supports the UML model and provides some limited integration within and between those views. OCL is a formal language for expressing constraints on model elements in UML. Since users can extend UML (e.g. through stereotypes), OCL can also help in integrating new techniques into UML. For instance, some work was done in integrating Architecture Description Languages, like C2 and Wright into UML [Medidovic 1998]. We will make use of OCL in further integrating architectural views in UML.

---

## 10.3 FROM INCEPTION TO TRANSITION

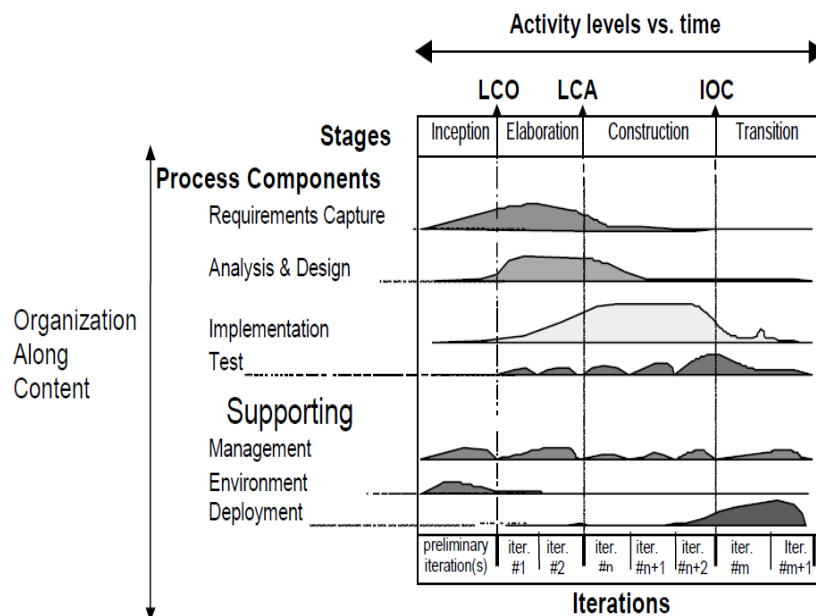
---

Naturally, a software development model should contain enough information so that it may be useful throughout the entire development life-cycle. Thus, the development model should have views giving information about the requirements, the organization, the people, the technology, the history of changes, the structure of the product, its transition, and so forth.

Seeing the model solely as a collection of views is however not right either. A development model should be more than that. For instance, it should give information on how to use the model and its views. As such, it needs to answer questions like ‘what has to be done first’ and ‘who needs to do what.’ Traditionally, models which give guidelines are commonly referred to as process models or life-cycle models and the probably best-known but also most controversial process model is the *Waterfall Model* [Royce 1970]. It defines the development process as a rather sequential process (feedback loops are allowed) with an upfront planning stage, followed by high level design and implementation. Even though it is widely acknowledged that

a purely sequential process is generally infeasible to follow, the stages it defines are still visible in many newer process models.

In the literature we often read that the Waterfall model is dead and clearly no other model has as of this date gained a similar popularity. Especially the object-oriented community seems to hold the belief that OO development is different and that regular process models do not seem to apply here. Even though the latter may be true, this does still not mean that process models are obsolete. The wisdom which was gathered and incorporated into the Waterfall model (or any other process model for that matter) was useful not because we used those process models but because we needed assistance with the intrinsic properties of the software “werewolf” – complexity, conformity, changeability, and invisibility. As it was mentioned above, the object-oriented paradigm may ease some of those problems but the fact remains that we are still dealing with the same problems we have dealt with 30 years ago – even with OO.



### Rational Unified Process with the WinWin Spiral Model's Anchor Points

We therefore can only conclude that the myth that we do not need a life-cycle (process) model because we are using OO design and/or programming languages [Siegfried, 1996] is exactly that – a myth. Siegfried states this very nicely when he says that “using object-oriented methods does **not** mean that we can ignore what we have learned since the 1950s.” Most of the lessons, best practices, guidelines, etc. which we have accumulated in the last 30 years are still as valuable and useful as they were before. It is just the way we have to use them which changed with the OO paradigm. Supporting a useful process model for OO development is therefore an essential step in integrating our OO views and there are a number of very good approaches. For instance, there is the WinWin Spiral Model [Boehm 1996] and the Rational Unified Process [Kruchten 1999]. Figure 4 shows an overview of the Rational Unified Process as it has integrated with the LCO, LCA, and IOC Anchor Points of the WinWin Spiral Model (defined later). It shows the major UML stages of an OO development process called *Inception*, *Elaboration*, *Construction*, and *Transition*. Each stage may use one or

more iterations (e.g. spiral cycle) to complete it and each milestone must deal to some degree with the process components listed on the left. For instance, initially, we focus most on requirements capture and analysis whereas later we concentrate more strongly on implementation and test. However, all of the activities proceed concurrently, as it can be seen in the figure. Even during construction new requirements may be identified, thus, the iterative nature of the life-cycle model. It is out of the scope of this work to go deeper into this subject. For more an overview on process models also refer to [Chroust 1992].

---

## 10.4 SOFTWARE ARCHITECTING

---

It is out of the scope of this work to incorporate all forms of development views. As mentioned above we will restrict ourselves mostly to UML views (which are explained in more detail later) and even there we will emphasize the high-level design and architectural views only. This section will therefore describe what we mean by the *Architecture* of a system.

To define what software architecture is, is already a problem in itself. Not many people can agree on a single definition. Thus, let me start by describing the common usage of the term which is based on the analogy to building architectures (an early definition is given by Wolf-Perry [1992]). The analogy between buildings and software may not be striking on first glance but the following may make it clearer.

Both buildings and software have an implementation representation; the actual building in bricks, stones, etc. and the implementation of the software product in a programming language. Both building and software architects use logical descriptions (blueprints) to describe the implementation. The building architect does so by describing the building's essential features without committing too much to specific construction details such as what kind of building materials to use. Similarly we would like to architect software without over committing to implementation details (e.g. what programming language to use). The building as such may stand for many years but may undergo some changes – superficial ones or more profound structural ones. In general, software has to undergo more changes throughout its life time – some more to the eye (GUI) others more profound. Both buildings and software have in common that some design decisions (which must be done early on) may have considerable impact on other decisions later on. For instance, a wall that needs to support a ceiling cannot easily be moved or removed (if at all). Similarly, some design decisions in software (e.g. commitments to COTS products) may impose similar constraints.

Furthermore, building architects are able to reuse some of their architectural information. For instance, if a settlement with a number of similar (not necessarily identical) houses is built, the architect may still be able to use most parts of the blueprint of a house for the others. In times where software development concepts such as product lines are gaining in popularity, we clearly would also like to have something equivalent to that for software architecting. And finally, a building architect is able to describe a building in sufficient detail so that construction workers can build it without too much interaction with the architect. Thus, the architectural description of the building is complete enough to minimize additional

mouth-to-mouth interaction between designers and builders – attributes we also would like to see in software architecture descriptions.

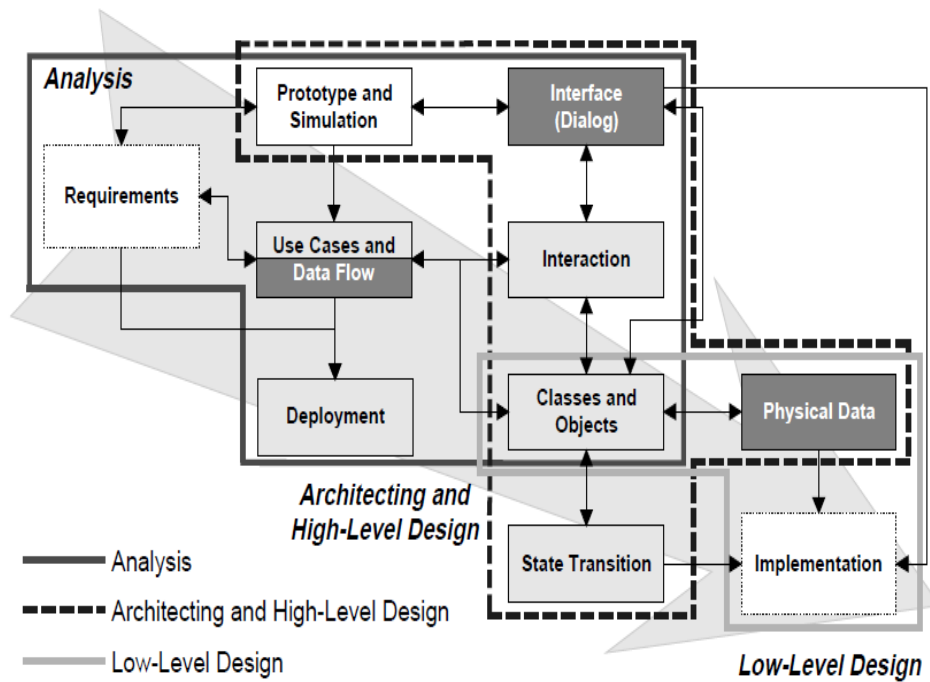
Clearly, software architecture and building architecture have many things in common, at least in principle. In practice, software architecture descriptions do not come anywhere close to the preciseness and general understandability of building architecture descriptions. However, in the long run we hope that we may yet come closer to that goal.

Recently, there has been some attempt in standardizing what architecting means. The IEEE Draft Standard 1471 is one of the latest. Let us provide their definition for architecture which illustrates some of the challenges in defining the term.

Every system has an architecture, defined as follows: An architecture is the highest-level conception of a system in its environment where: the ‘highest-level’ abstracts away from details of design, implementation and operation of the system to focus on the system’s ‘unifying or coherent form’; ‘conception’ emphasizes its nature as a human abstraction, not to be confused with its concrete representation in a document, product or other artifact; and ‘in its environment’ acknowledges that since systems inhabit their environment, a system’s architecture reflects that environment.

This definition’s primary problem is vagueness: it could apply equally well to “architecture,” “requirements,” or “operational concept.” Another problem we see with this definition is that it seems to emphasize too little onto the analysis and interpretation of architectural descriptions. We *architect* not only because we want to build but also because we want to understand. Thus, architecting has a lot to do with analyzing and verifying the conceptual integrity, consistency, and completeness of a design. This does not mean that above description excludes the analysis of architectures. Quite contrary. Everybody acknowledges its existence as part of architecting. However, it seems that builders of architectural descriptions only think about their analysis as a second thought and then mostly in vague terms. UML is certainly a very good example of that. This deficiency will be investigated shortly.

Architecting in UML may be seen in Figure. This figure shows UML views and other views which often are part of the major development stages (from the developers point of view) – the analysis, architecture, and design of a software system. The arrows depict the dependencies of the views onto information in other views. This figure should not be taken too literally since we tried to capture the major flows of dependencies only. For instance, the picture shows that the classes and objects affect the implementation (e.g. code in C++) but not vice versa. This is, of course, not always true.



## 10.5 ARCHITECTURAL VIEWS IN UML

There are cases where the implementation may trigger changes in the architecture (e.g. due to choice of COTS product). As a general rule, it is good practice to anticipate these dependencies and address them via prototyping and analysis as emphasized in Figure.

Further, the associations of the development artifacts (such as classes, use cases, etc.) to the major phases of the life cycle can indicate primary associations at best. Again, we tried to capture the major associations of those development artifacts and the views in which they are frequently used. It is this ambiguity in how to associate and relate those artifacts which already poses our first problem in OO development. Traditional life-cycle models such as the Waterfall Model are less useful in OO development because of the activity and artifact overlaps we discussed in Figure . This fact can also be seen in Figure where some development artifacts, such as classes and objects, are used and shared extensively during the entire development process. This ambiguity, in the definition of development stages and phases, is however also a good thing since it provides some continuity between the life-cycle stages and, thus, brings the development stages closer together. The conceptual breaks, which so frequently happen between the analysis and design stages, are eased.



## ARCHITECTURAL VIEW

### Unit Structure

- 11.1 Architectural views
- 11.2 Logical Architecture
- 11.3 Subsystems
- 11.4 Hardware Architecture
- 11.5 Process Architecture
- 11.6 Implementation Architecture

---

### 11.1 ARCHITECTURAL VIEWS

---

An architectural view is a category of diagrams addressing a specific set of concerns. All the different architectural views determine the different ways in which we can understand a system. For example, all the figures shown so far in this book may be organized into different views, including those for addressing functional, structural, behavioral, and other pieces of the project management system. The elements that make up a view are known as view elements. For example, all the elements in the figures are view elements when we classify the diagram on which they appear into a specific view.

Because the UML is a language and not a methodology, it does not prescribe any explicit architectural views, but the UML diagrams may be generally organized around the following commonly used architectural views:

---

### 11.2 LOGICAL ARCHITECTURE

---

Focuses on the logical view of the system using class diagrams, dependency, class visibility and sub systems to communicate the existence of the classes and their relationship.

#### Dependencies

A dependency from a source element ( called the client) to a target element (called the supplier) indicates that the source element uses or depends on the target element; if the target element changes, the source element may require a change.

A dependency is shown as a dashed-line path from the source element to the target element. The dependency may be marked with the use keyword; however, the keyword is often omitted because the meaning is evident from how the dependency is used. Also, notice that a dependency does not have a large hollow triangle at the end of the path, but has an open arrow.

## Class Visibility

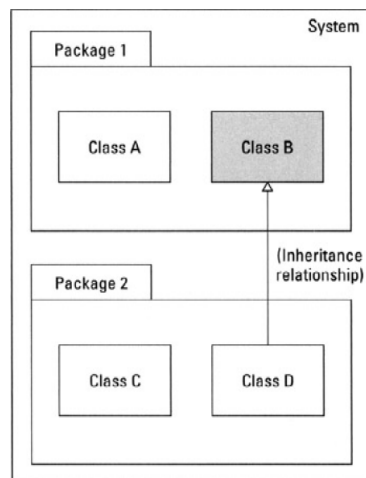
- 1) scope of access allowed to a member of a class
- 2) applies to attributes and operations

UML visibility maps to OO visibilities:

- 1) private scope: within a class (-)
- 2) package scope: within a package (~)
- 3) public scope: within a system (+)
- 4) protected scope: within an inheritance tree (#)

## Private Visibility

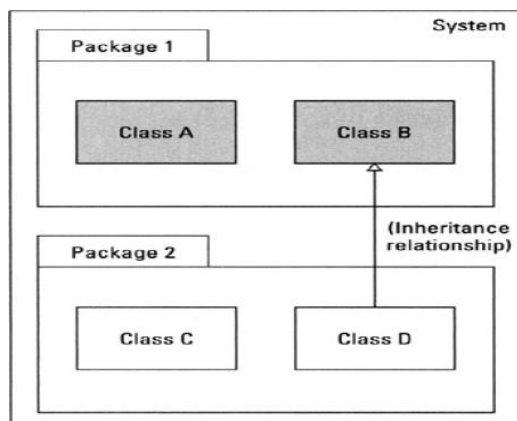
- 1) private element is only visible inside the namespace that owns it
- 2) notation is “-”
- 3) useful in encapsulation.



## Package Visibility

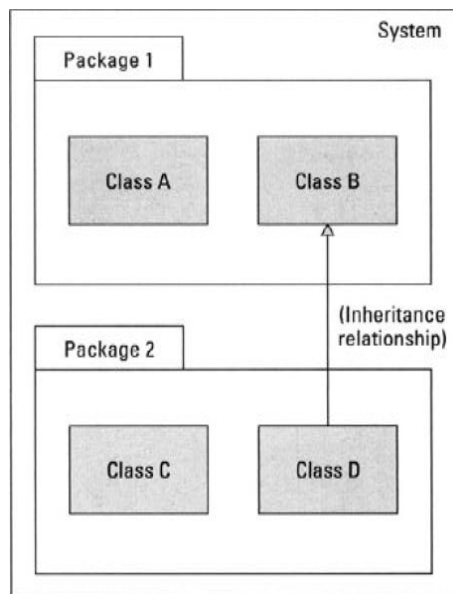
- 1) package element is owned by a namespace that is not a package, and is visible to elements that are in the same package as its owning namespace

- 2) notation is “~”



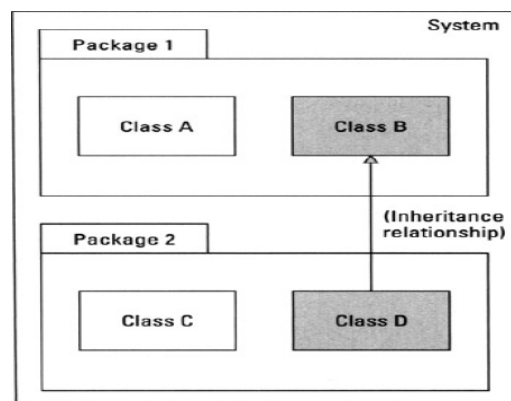
## Public Visibility

- 1) public element is visible to all elements that can access the contents of the namespace that owns it
- 2) notation is “+”



### Protected Visibility

- 1) a protected element is visible to elements that are in the generalization relationship to the namespace that owns it
- 2) notation is “#”




---

## 11.3 SUBSYSTEMS

---

Recall that a system is an organized collection of elements that may be recursively decomposed into smaller subsystems and eventually into nondecomposable primitive elements. For example, the project management system may be decomposed into the following:

- A user interface subsystem responsible for providing a user interface through which users may interact with the system
- A business processing subsystem responsible for implementing business functionality
- A data subsystem responsible for implementing data storage functionality

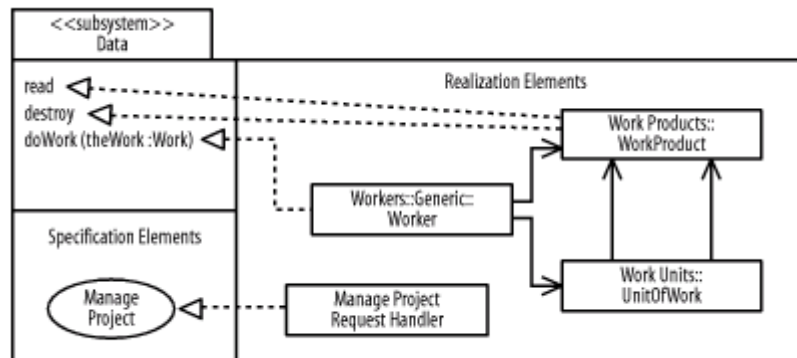
The primitive elements would be the various classes that are used in these subsystems and ultimately in the whole system. While a package simply groups elements, a subsystem groups elements that together provide services such that other elements may access only those services and none of the elements themselves. And while packages allow us to partition our system into logical groups and relate



these logical groups, subsystems allow us to consider what services these logical groups provide to one another.

A subsystem is shown as a package marked with the `subsystem` keyword. The large package rectangle may have three standard compartments shown by dividing the rectangle with a vertical line and then dividing the area to the left of this line into two compartments with a horizontal line. Following figure shows how a Data subsystem for our project management system might look. The subsystem's operations, specification elements, and interfaces describe the services the subsystem provides, and are the only services accessible by other elements outside the subsystem.

#### A subsystem's representation in the UML



The upper-left compartment shows a list of operations that the subsystem realizes. The lower-left compartment may be labeled "Specification Elements" and shows specification elements that the subsystem realizes. For example, any use cases that the subsystem provides are specification elements that the subsystem must realize. The right compartment may be labeled "Realization Elements" and shows elements inside the subsystem that realize the subsystem's operations and specification elements as well as any interfaces that the subsystem provides. You can modify this general notation by rearranging compartments, combining compartments, or completely suppressing one or more compartments. Any element may be used as a specification or realization element, because a realization simply indicates that the realization element supports at least all the operations of the specification element without necessarily having to support any attributes or associations of the specification element.

---

## 11.4 HARDWARE ARCHITECTURE

---

Focuses on the implementation environment, using deployment diagrams to communicate how the implemented system resides in its environment.

### Deployment Diagram

**Introduction :** Here we focus deployment diagrams, which depict the implementation and environment of a system, respectively. First, we introduce deployment diagrams and how they are used.

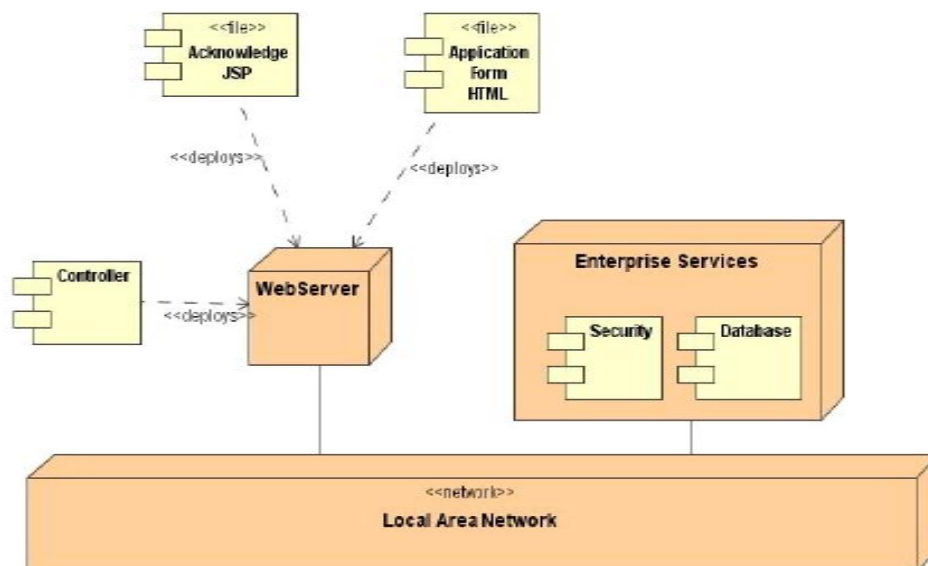
Deployment modeling is a specialized type of structural modeling concerned with modeling the implementation environment of a system. In contrast to modeling the components of a system, a deployment

model shows you the external resources that those components require. You typically apply deployment modeling during design activities to determine how deployment activities will make the system available to its users; that is, to determine the elements of the system on which deployment activities will focus. Like component modeling, deployment modeling usually starts after the design of the system is fairly complete, as determined by your system development process.

## Deployment

- 1) shows configuration of run-time processing nodes and the components hosted on them
- 2) addresses the static deployment view of an architecture
- 3) is related to component diagram with nodes hosting one or more components
- 4) essentially focus on a system's nodes, and include:
  - a) nodes
  - b) dependencies and associations relationships
  - c) components
  - d) packages

For e.g.



## **Nodes and Components**

### **Components**

- 1) participate in the execution of a system.
- 2) represent the physical packaging of otherwise logical elements

### **Nodes**

- 1) execute components
- 2) represent the physical deployment of components

The relationship between a node and a component can be shown using a dependency relationship.

### **Organizing Nodes**

Nodes can be organized:

- 1) in the same manner as classes and components
- 2) by specifying dependency, generalization, association, aggregation, and realization relationships among them.

The most common kind of relationship used among nodes is an association representing a physical connection among them.

### **Processors and Devices**

A processor is a node that has processing capability. It can execute a component.

A device is a node that has no processing capability (at least at the level of abstraction showed).

### **Modelling Nodes**

#### **Procedure:**

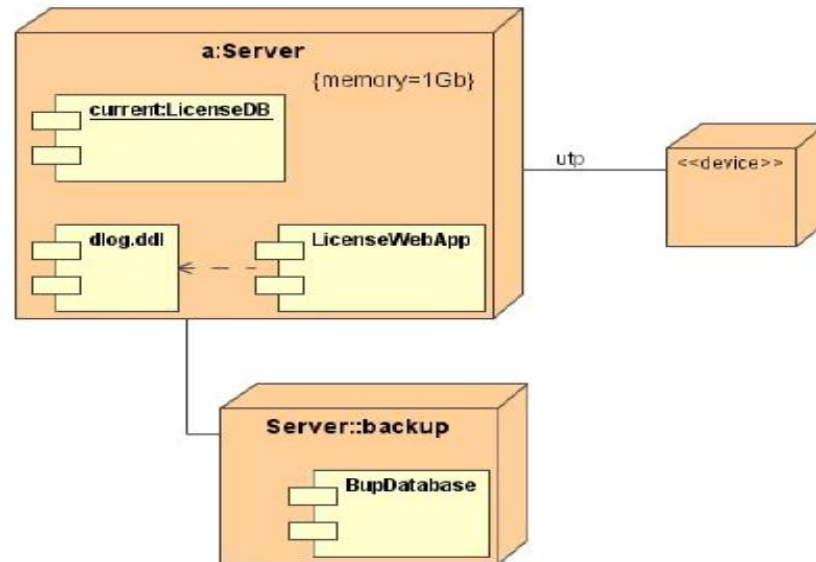
- 1) identify the computational elements of the system's deployment view and model each as a node
- 2) add the corresponding stereotype to the nodes
- 3) consider attributes and operations that might apply to each node.

### **Distribution of Components**

To model the topology of a system it is necessary to specify the physical distribution of its components across the processors and devices of the system.

#### **Procedure:**

- 1) allocate each component in a given node
- 2) consider duplicate locations for components, if it is necessary
- 3) render the allocation in one of these ways:
  - a) don't make visible the allocation
  - b) use dependency relationship between the node and the component it's deploy
  - c) list the components deployed on a node in an additional compartment



## 11.5 PROCESS ARCHITECTURE

Focuses on the process and threads and their notations in UML to communicate

## 11.6 IMPLEMENTATION ARCHITECTURE

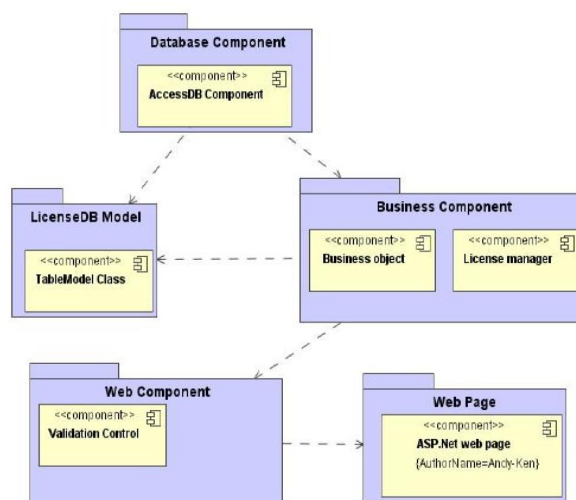
Focuses on the implementation of a system, using component diagrams to communicate how the system is implemented.

### Component Diagram

It shows structural replaceable parts of the system. Its main components are:

- 1) components
- 2) interfaces
- 3) packages

For example.



## Components

Definition : A component is a physical, replaceable part that conforms to and provides the realization of a set of interfaces.

A component:

- 1) encapsulates the implementation of classifiers residing in it
- 2) does not have its own features, but serves as a mere container for its elements
- 3) are replaceable or substitutable parts of a system

For e.g. following is a component named OrderEntry.exe.



## Interface

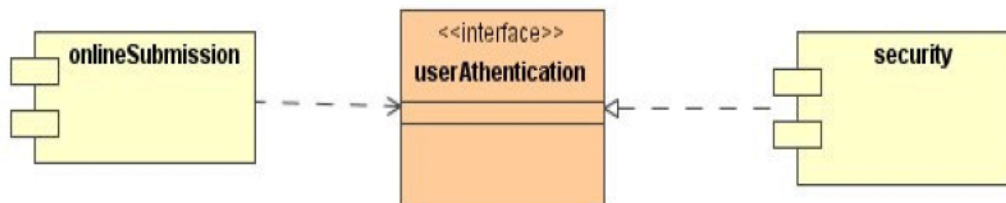
Definition : An interface is a collection of operations that are used to specify a service of class or components.

Interfaces:

- 1) represent the major seam of the system
- 2) are realized by components in implementation
- 3) promotes the deployment of systems whose services are location independent and replaceable

Relationship between a component and its interface may be shown as follows,

- a) interface is presented in an enlarged form, possibly revealing operations elided iconic form
- b) realizing component is connected to it using a full realization relationship



## Packages

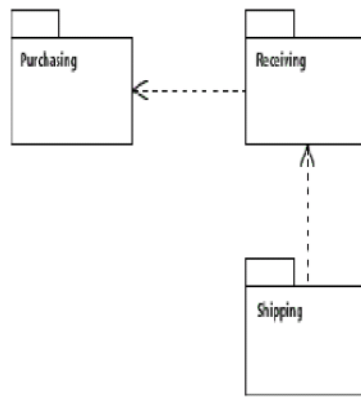
Packages:

- 1) are general purpose mechanism for organizing modeling elements into groups
- 2) group elements that are semantically close and that tend to change together

Packages should be loosely coupled, highly cohesive, with controlled access to its contents

## Package Notation

- drawn as a tabbed folder
- packages references one another using standard dependency notation
- for instance: Purchasing package depends on Receiving package
- packages and their dependencies may be stereotyped.



## Nodes

A node is a resource that is available during execution time. Traditionally, nodes refer to computers on a network, but in the UML a node may be a computer, printer, server, Internet, or any other kind of resource available to components.

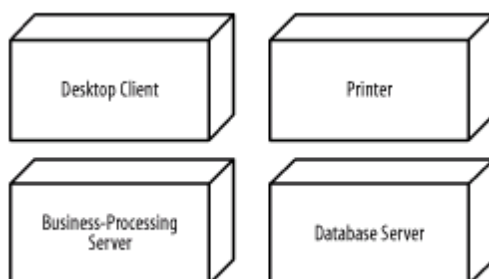
For example, the project management system may be deployed on the following nodes:

- **A desktop client** : On which the user interface component executes
- **A printer** : Which the project management system uses to print reports
- **A business-processing server** : On which the business-processing component executes
- **A database server** : On which the data component executes and where project-related information is stored

When speaking of a class of nodes, it's customary to use the terms node or node class. Thus, while you might think of a node as a specific thing, in the UML, a node really represents a class of nodes. When speaking of a specific component of a class, use the term node instance.

A node is available during execution time and is a resource on which components may execute. In the UML, a node is shown as a three-dimensional rectangle labeled with the node's name.

Following figure shows various nodes associated with the project management system, including a desktop client, business-processing server, database server, and printer node.



## VIEW INTEGRATION

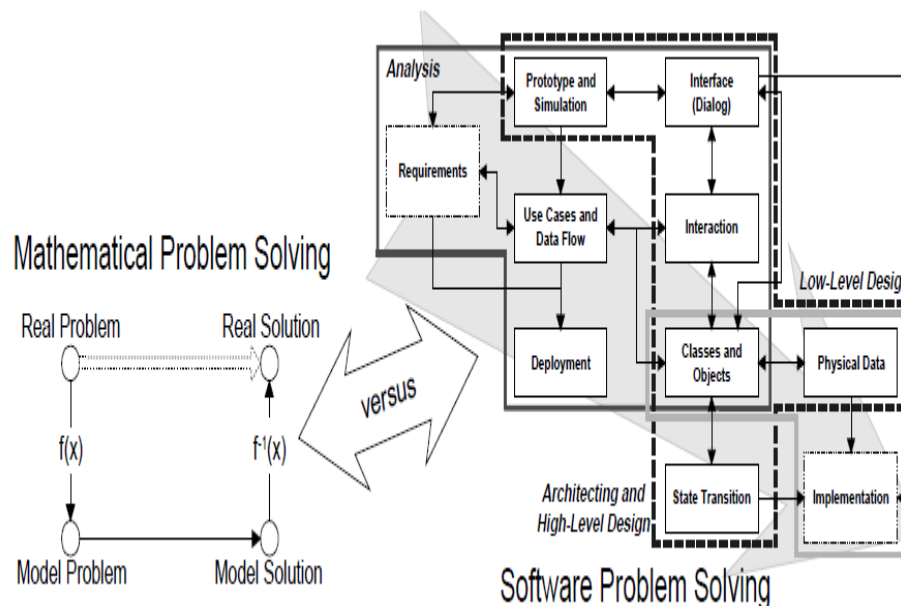
### Unit Structure

- 12.1 Missing Integration in Models and Views
- 12.2 What is Integration?
- 12.3 The View Integration Problem
- 12.4 Why Integrate Architectural Views?
- 12.5 Why Integrate Architectural Views in UML?

### 12.1 MISSING INTEGRATION IN MODELS AND VIEWS

In the previous chapter we defined what it means to do OO software development. We talked about models and views, and used UML as an example. We confirmed the continuous need for a life-cycle model to guide us through the development process and we talked about architecting as being one of the most critical development stages from the software developer's point of view.

With that we basically described everything a well-equipped development team has access to these days. However, we still have a major problem. When we described the mathematical problem solving approach we concluded that modeling (architecting) replaces the finding of a solution to the real problem by the finding a solution to the model problem. For that very reason, software development models were devised which serve as counterparts to the mathematical model. However, are our software engineering models really equivalent to the mathematical model in solving the problem? What if the (software) model which we created to represent the real world is not adequate? A solution we might find to that model problem would therefore not be correct. This implies that we are not only confronted with the challenge of finding a (model) solution to a model problem but also to find a



## Two Problem Solving Approaches

model of the real world which adequately represents it for our needs. This is like solving the right problem vs. solving the problem right! As such the Mathematical Problem Solving Approach is really doing three things (corresponding to the three arrows):

- **Model the real problem adequately**
- **Solve the model problem**
- **Interpret the model solution in the real world**

Which part of the software model in Figure is doing the modeling? Which part is doing the solving? And which part is doing the interpretation of the solution? None? But, what does this tell us about conventional software development models such as the UML? For instance, what is the best implementation of a software product if it does not reflect the architecture? What is the best architecture if it does not satisfy the requirements?

The only conclusion we can draw from that case is that Architecting is more than what conventional development models provide. ***Architecting is to model, to solve, and to interpret.*** And techniques such as the ones used by UML are just providing assistance. Therefore, this work is about integrating views so that they provide more than just structural assistance. In particular, we will investigate the integration of some architectural views in UML and what techniques we can deploy to bridge the gap between what architectural models are and what they should be.

---

## 12.2 WHAT IS INTEGRATION?

---

We have used the word *Integration* or ‘what it means to *integrate*’ but so far we have not described it. This section will do that. The term *Integration*, as such, is part of everybody’s vocabulary. Therefore, let us check how the Merriam-Webster Dictionary defines that term. There, *Integration* is defined as:

- 1) The act or process or an instance of integrating: as a) incorporation as equals into society or an organization of individuals of different groups (as races) b) coordination of mental processes into a normal effective personality or with the individual’s environment
- 2) The operation of finding a function whose differential is known; the operation of solving a differential equation

This set of definitions for the term *Integration* is very general. That should, however, not surprise us since this is how the word *Integration* is used. In Software Engineering it applies to Technology, Organization, and People; it affects management, products, humans, politics, standards, models, enterprises, and much more. Sage and Lynch’s work about *Systems Integration and Architecting* [Sage-Lynch 1998] provide a very comprehensive overview of what integration in our context means. They found that “Systems Integration is an activity omnipresent in almost all of systems engineering and management.” They further found that “the term lacks precise definition and is used in different ways and for different purposes in the engineering of systems.”



In software engineering and software architecting, the word *Integration* is used frequently. It often refers to the process of assembling components (or subsystems) into a system. As such, the term integration stands for an activity that starts later on in the life-cycle once some components of the software system are developed. Another case, where the term *Integration* is used, refers to the unification of standards, processes, and models. For instance, the Integrated Capability Maturity Model (iCMM) of the FAA (which is a union of various CMM models (such as the SW-CMM [Paulk 1995], SE-CMM [Kuhn 1996], SACMM [Ferguson et al., 1996] and so forth) is one such attempt to combine standards to a more general one. The Unified Modeling Language (UML) is another such case, where various object-oriented development models (Booch, OMT, and pieces of many others) were combined into a integrated single OO development model.

In this work, the term *Integration* is used, in yet another way, to determine the semantic integrity of development models (or views, diagrams, etc.) in order to evaluate or improve quality aspects of the development model. Desirable qualities we would like to see in a development model (or its instances such as the product or domain models) are consistency, completeness, correctness, and so forth. So we may ask ourselves:

- What does it mean, for one view to be *consistent* with another view?
- When do I know, whether one view presents a *complete* picture of the entire system?
- How do I know that what I did is correct and faithfully represents what my customer wanted me to do?

On a close look, this form of *integration* is, however, not very different from the meaning described above. For instance, when we perform a component integration where we evaluate the integrity of components while assembling them to a bigger component (or even system) this is quite analogous to performing a view integration where we evaluate the integrity of views while assembling them to a bigger model. The one describes the product integration, the other the view integration. Both are facets of ***Integration*** (see also [Grady 1994] for an overview of these facets).

---

## 12.3 THE VIEW INTEGRATION PROBLEM

---

### Why Integrate Views?

Above we briefly described an object-oriented development model (UML) which satisfies the need of our stakeholders (such as users, designer, programmers, and maintainers) for views which can be used by them to describe and communicate. We further briefly described a process which can be used to guide them and advise them on how to use those views in creating a useful and feasible software product. And we also described the deficiency of that approach when it comes to solving the problem (to model, to solve, and to interpret).

This deficiency in views would not exist if we could have a few *perfect views* that could be used by all stakeholders (as described above) and which were precise enough but still easy enough to use.

These views, unfortunately, do not exist. Instead, we are confronted with a number of loosely coupled, sometimes quite independent views. This is not really what we want. [Nuseibeh 1996] wrote that “multiple views often lead to inconsistencies between these views – particularly if these views represent, say, different stakeholder perspectives or alternative design solutions.”

Thus, if we have to deal with multiple views we would like to have at least tightly coupled ones. Since a view represents only one aspect of the system to be modeled, they are meant to be together – only together can they fully describe the system. However, we also need those views to be different (and independent) enough to provide useful meaning to their respective stakeholders. Therefore, what we need are views which are independent and can stand on their own, but with their contents being fully integrated with the contents of the other views to ensure their conceptual integrity. Thus, we need *View Integration*.

We also need integration, because the views often use different underlying paradigms and, thus, the results of modeling a system in one view may be different than modeling the same system in another view. For instance, a non object-oriented analysis and design stage would yield functional model elements as its major components (which are more suitable to be implemented in a functional programming language). Instead, using object-oriented design techniques (or views) would already structure the system in a more object-centered fashion and thus, its implementation will be more straightforward in an OO language.

In Figure we showed both object oriented (classes, interaction) and functional views (data flow, state transition) which are commonly used together in OO development. So if two different people would start creating a system, one using OO techniques and the other using functional ones, we would most likely get two different solution approaches for the same system. Even if each view were correctly solving the problem, they would still not make much sense together. This is because the one type of technique would yield a system which is structured by its functions whereas the other one would be structured by objects which have behavior (functions). Further, if modeling is done separately (one view at a time) we may get inconsistencies between them. The notation described above does not describe the semantics of the model and how it is (or is not) supposed to be used. Life-cycle process descriptions may help in that but they are usually not detailed enough and for the most part they are not supported by tool.

Thus, what we need is a development model which is not only defined syntactically but also semantically. Such a model would also need tool support which would not only enable the architects to create a model instance for a system which is syntactically correct but the tool should also be able to verify the semantic integrity of the model instance (at least to some degree).

The integration of architectural views (as the title says) is about adding semantics to our architectural views so that the integrity of the whole is improved.

---

## 12.4 WHY INTEGRATE ARCHITECTURAL VIEWS?

---

The reason why we chose the integration of architectures was because it is the most important part of the design. This is best explained by [Siegfried 1996] who wrote that “there is no replacement for making a sound systems architecture early in a project.” Architecting is the start of a development process from a pure engineering point of view. Architecting is also early in the development life-cycle which means that problems and faults are still relatively easy (and inexpensive) to fix. Should architectural errors be carried into the implementation phase or even further, the cost of fixing them are some orders of magnitude higher [Boehm 1981]. Further, architectural descriptions are already low-level enough to be less ambiguous. Thus, there is more precise information available from which we can draw from. Integrating requirements (which is equally important) is much harder to do since we would require techniques such as natural language understanding which do not exist in a level of sophistication suitable for our needs.

---

## 12.5 WHY INTEGRATE ARCHITECTURAL VIEWS IN UML?

---

The beginning of this work already indicated why we chose object-oriented technology as another cornerstone of this work. OO is more and more dominating the market and UML has evolved into the most significant OO analysis and design methodology. The Object Management Group (OMG), furthermore, has standardized UML for the Object Management Architecture (OMA). “The adoption of UML provides system architects working on Object Analysis and Design with one consistent language for specifying, visualizing, constructing and documenting the artifacts of software systems, as well as for business modeling.” [OMG 1997] We also chose UML because others have already made some progress in integrating UML views. Thus, summarizing previous sections we can say that views alone are not solving the consistent architecture representation problem because they:

- are standalone/independent
- involve different types of modeling elements
- are for different audiences/stakeholders
- are often used concurrently

This means that same or similar information is entered multiple times and that related information must be kept consistent manually. The *View Integration Problem* exists because it is often not apparent what information is duplicated or inconsistent. Therefore, finding means of ensuring the conceptual integrity are based on the ability of identifying duplicated model elements and integrating their properties.

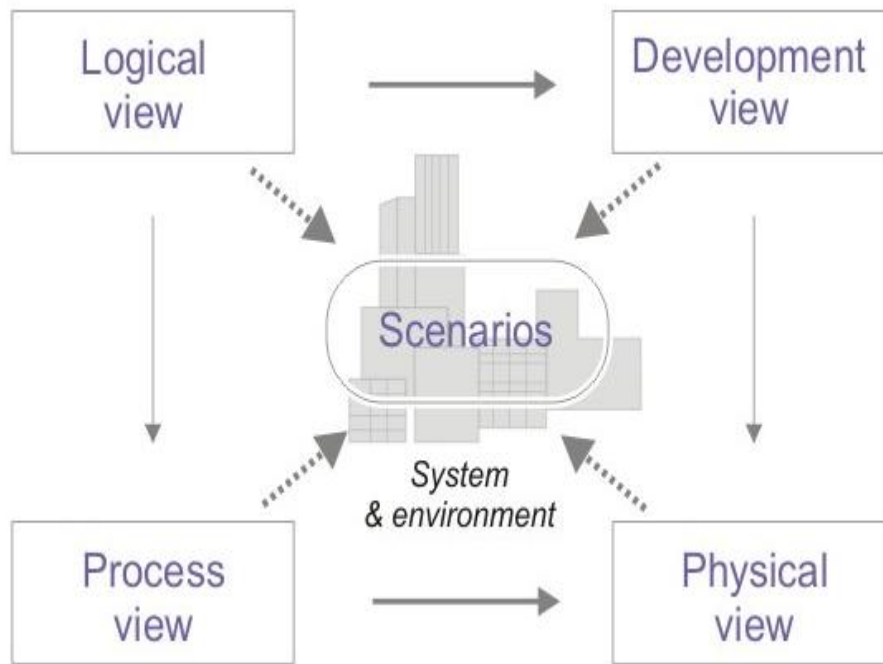


# ARCHITECTURE VIEW MODEL

## Unit Structure

- 13.1 Architectural View Model
- 13.2 Dimensions of View Integration
- 13.3 Types of Views
- 13.4 Flexibility Guidelines for Class Design
- 13.5 Flexibility Guidelines for Behavioral Design

### 13.1 ARCHITECTURAL VIEW MODEL



Architecture view model designed by Philippe Kruchten for "describing the architecture of software-intensive systems, based on the use of multiple, concurrent views". The views are used to describe the system from the viewpoint of different stakeholders, such as end-users, developers and project managers. The four views of the model are logical, development, process and physical view. In addition selected or scenarios are utilized to illustrate the architecture serving as the 'plus one' view. Hence the model contains views:

- **Logical view** : The logical view is concerned with the functionality that the system provides to end-users. UML Diagrams used to represent the logical view include, Class diagram, Communication diagram Sequence diagram.
- **Development view** : The development view illustrates a system from a programmer's perspective and is concerned with software management. This view is also known as the implementation view. It uses the UML Component diagram to describe system components. UML Diagrams used to represent the development view include the Package diagram.

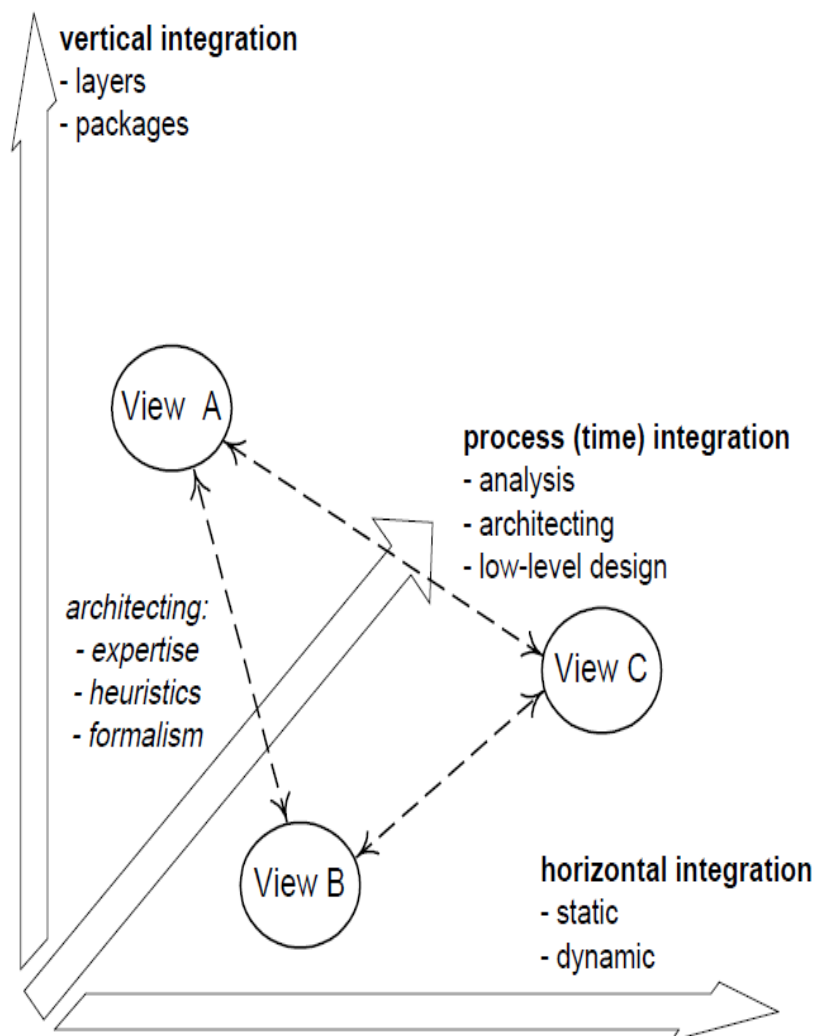
- **Process view** : The process view deals with the dynamic aspects of the system, explains the system processes and how they communicate, and focuses on the runtime behavior of the system. The process view addresses concurrency, distribution, integrators, performance, and scalability, etc. UML Diagrams to represent process view include the Activity diagram.
- **Physical view** : The physical view depicts the system from a system engineer's point-of-view. It is concerned with the topology of software components on the physical layer, as well as the physical connections between these components. This view is also known as the deployment view. UML Diagrams used to represent physical view include the Deployment diagram.
- **Scenarios** : The description of an architecture is illustrated using a small set of use cases, or scenarios which become a fifth view. The scenarios describe sequences of interactions between objects, and between processes. They are used to identify architectural elements and to illustrate and validate the architecture design. They also serve as a starting point for tests of an architecture prototype. UML Diagram(s) used to represent the scenario view include the case diagram..

---

## 13.2 DIMENSIONS OF VIEW INTEGRATION

---

To get a better understanding of how views fit together, consider Figure. For our purposes we divide views into three major dimensions; horizontal, vertical and process. The vertical and horizontal dimensions are also often referred to as layers and partitions whereas the process dimension reflects the life-cycle (time). These three dimensions are briefly described in the following. It is worth noting that these dimensions are also three



**Figure : Dimensions of Views**

view integration dimensions since we would like to integrate views within their various dimensions but also across dimensions.

### **Vertical View Dimension**

The vertical dimension reflects the system decomposition. Usually, higher levels (layers) show the system in a more abstract fashion whereas lower levels show the system in more detail. Each layer should represent a *complete* system, however, it might not do so in one diagram. Mismatches between vertical views are therefore mismatches where one layer does not represent the same interrelationships (within and between diagrams) as another layer. In UML, system decomposition is primarily achieved through class/object diagrams and associated packages. Other diagrams, such as state diagrams do not reflect the system decomposition. They may, however, still be used in each layer to repeat the same modeling constructs in an increasingly detailed and abstracted form.

Once a system is decomposed into subsystems, additional refinements (layers) of that subsystem must, then, not reflect the entire system any more but instead only that particular subsystem. If this is the case, other subsystems have to be similarly represented by their own layers and all those (sub) layers together should represent the complete system again. Thus, there may be another category of mismatches between (sub) layers if the partitions of those layers are not fully separated and consistently applied. In some cases, subsystems may not

need further refinements because they already are detailed enough whereas in other cases subsystems (layers) may still be in need of further refinements. If this happens, the already adequately refinement subsystem(s) (which do not need further refinements) may also be used (e.g. accesses, called, etc.) by all subsequent refinements of other lower layers so that the entirety may still represent a complete system. This case also shows that the level of detail achieved by a layering is often not clearly defined and it may vary within each layer. The only important aspect is whether the sum of its part represents a complete picture again and that the logical divisions (partitions) of the system into subsystems is reflected consistently on all levels of abstractions.

There are however variations in how this can be done. For instance, the physical design should be a continuation of the logical design. This does however not mean that physical layers have to use the same type of views or the same set of features of logical layer. For instance, the physical design (as compared to the logical one) should only use modeling elements that can be directly implemented. So, if a system layer is physically described in a class diagram and it is going to be implemented in C (a non-OO language) than some elements of class diagrams, such as inheritance, should not be used in the physical design anymore. Nevertheless, the partitions into which subsystems are divided are still valid and should be the same for both logical and physical views. Thus, it is not required that layers must use the same type of view (e.g. class diagrams).

### **Horizontal View Dimension**

The only restriction we have from the vertical integration is that the set of views used in each layer must represent the entire system completely. However, each layer may still be represented by different sets of views. It may do so using different types views or even different decompositions (although the latter is not recommended as discussed before). Thus, in the horizontal layer, a view is not required to model a complete system (or subsystem depending on the layer).

Horizontal views are frequently further divided into static and dynamic views. The difference of these two groups is related to the presumed execution time. Dynamic views represent the system (or more likely a partition of it) at a particular point of time or time interval. For instance, object diagrams show the objects that exist at a particular time; sequence diagram show the calling dependencies between various objects during a time interval. Dynamic views often represent samples of the state or interaction of the system and its components during their execution. Class diagrams on the other hand are static representations of all allowed dependencies of a system and its components throughout the execution. Views in that category represent more general aspects of the system behavior and their constructs are always applicable.

Most forms of representations (diagrammatic or not) can be used in this horizontal manner. This fact has made some people belief that architectures are 'flat'. However, this is, as Philippe Kruchten said during his keynote address at the GSAW'98, one of the ten major misconceptions about software architectures [Kruchten 1998].

## Process View Integration

Although, this form of integration is often ignored, it is actually very important. The integration over time (or process integration) reflects the integration of product artifacts during the life cycle. Note that we are not speaking of making sure that a process model is followed consistently but to make sure that the changes a product artifact goes through over time are captured. This activity is also often referred to as version control or configuration management but it is also another dimension of the view integration problem. If we loose the rationale why things happen the way they did (or changed the way they did) we loose some important design information.

For instance, if we have two alternative design approach, each with unique advantages and disadvantages, clearly, we would like to make sure that both alternatives are reflecting the same problem and do so completely. So it may make sense to compare those two design approaches, and if it is only to ensure completeness. This integration aspect is, however, only of secondary importance to our work. We nevertheless list it because the activities and techniques presented in this work apply to this form of integration, as well.

---

### 13.3 TYPES OF VIEWS

---

Currently this work makes use of the following types of views. For a more detailed description please refer to the UML notation and semantics guide [Booch-Jacobson-Rumbaugh, 1997]. Later on in this work, we will expand on this list and show in a high level fashion how other architectural representations can be included as well (e.g. ADLs).

#### **Diagrammatic views:**

- Class/Object Diagrams
- Sequence Diagrams
- State Diagrams
- Interface Diagrams
- Collaboration Diagrams

#### **Textual views:**

- Object Constraint Language (OCL)
- Programming languages (C++)
- Attributes and other properties of UML model elements

---

### 13.4 FLEXIBILITY GUIDELINES FOR CLASS DESIGN

---

**Flexibility Guidelines for Class Design:** Guidelines and heuristics that lead to more extensible and reusable class designs. Coupling and cohesion in object-oriented programs. Class normalization for cohesion. Guidelines for the use of inheritance. Using aggregation versus using inheritance.

**UML Extension Mechanisms:** The use of properties, constraints, and stereotypes to extend the UML notation. Standard stereotypes, including interfaces. The use of interfaces to define roles.

**Concepts and Notation for Interaction Diagrams:** The concepts and notations for collaboration and sequence diagrams. Denoting iteration,



branching, and object creation and destruction in each type of diagram. The relationship of interaction diagrams to the class diagram.

**Concepts and Notation for State Transition Diagrams:** When to use state transition diagrams. The notation for the diagrams, including composite states, history states, and concurrent state machines.

**Behavioral Design Approaches:** The 'top-down' versus the 'bottom-up' approaches to designing class behaviors. Use cases revisited: three approaches for identifying a problem's use cases. The top-down process of identifying required scenarios, then turning those scenarios into interaction diagrams, object methods, and state machines. The bottom-up approach of concentrating on class responsibilities.

---

## 13.5 FLEXIBILITY GUIDELINES FOR BEHAVIORAL DESIGN

---

**Flexibility Guidelines for Behavioral Design:** Guidelines for allocating and designing behaviors that lead to more flexible designs. Coupling revisited. Avoiding centralized control. The overuse of accessor methods. Trading off extensibility versus reuse.

**System Architecture:** Layered architectures. The package concept and its UML notation. Defining layers and subsystems as packages. How to decompose a system into subsystems. The UML component diagram.

**Concurrency and Synchronization:** Threads and processes. Managing concurrent access to objects. Scheduling approaches. Introducing concurrency in UML interaction diagrams.

**Distribution and Persistence:** Physical distribution and the UML deployment diagram. Superimposing distribution on top of UML interaction diagrams. An introduction to object request brokers. Flat files versus relational databases versus object-oriented databases for persistence.

**An Introduction to Java:** A short introduction to the Java programming language as an example of an object-oriented programming language.

**Low-Level Design Idioms:** Guidelines for designing class interfaces. Idioms for the low-level design of class attributes, associations, and operations.



## REUSE : LIBRARIES, FRAME WORKS COMPONENTS AND PATTERNS-I

### Unit Structure

- 14.1 Reuse
- 14.1 Types of reuse
- 14.2 Introduction
- 14.3 Object-Oriented Programming
- 14.4 Abstract Classes
- 14.5 Software Reuse
- 14.6 Toolkits and Frameworks
- 14.7 White-box vs. Black-box Frameworks
- 14.8 Toolkits
- 14.9 Lifecycle

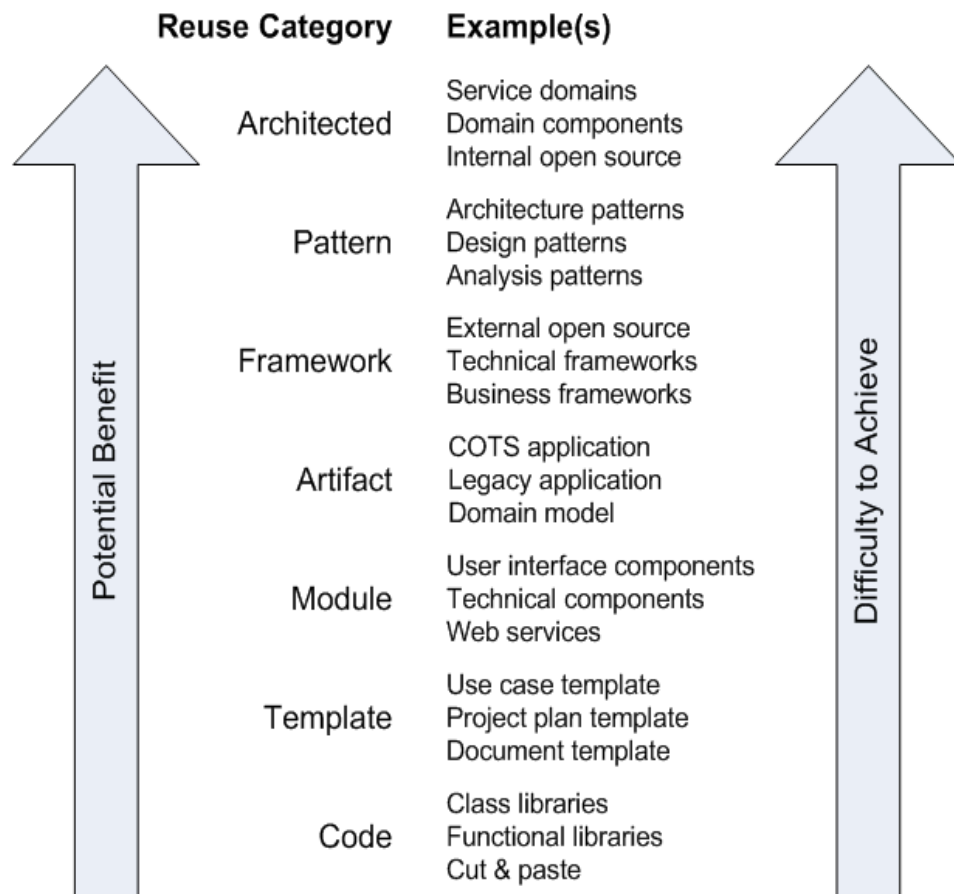
---

### 14.1 REUSE

---

Reuse is often described as not “reinventing the wheel” and the first step at succeeding at reuse is to understand that you have more than one option at your disposal. You can reuse source code, components, development artifacts, patterns, and templates. Figure 1, modified from A Realistic Look at OO Reuse, shows that there are several categories of reuse. The left-hand arrow indicates the relative effectiveness of each category – pattern reuse is generally more productive than artifact and framework reuse for example. Similarly, the right hand arrow indicates the relative difficulty of succeeding at each type of reuse. Code and template reuse are relatively easy to achieve because you simply need to find the asset and work with it. Other forms of reuse become hard to achieve; with framework reuse you need to learn the frameworks, with pattern reuse you must learn when to (and when not to) apply various patterns, and with architected reuse you need an effective approach to enterprise architecture in place.

Figure 1. Comparing types of reuse.



## 14.1 TYPES OF REUSE

- **Architected Reuse.** The identification, development, and support of large-scale, reusable assets via enterprise architecture. Your enterprise architecture may define reusable domain components, collections of related domain/business classes that work together to support a cohesive set of responsibilities, or service domains which bundle a cohesive collection of services together.
- **Pattern Reuse.** The use of documented approaches to solving common problems within a specified context. With pattern reuse you're not reusing code, instead you are reusing the thinking that goes behind the code. Patterns can be a multiple levels - analysis, design, and architecture are the most common. Ward Cunningham's site is a useful source of patterns on the web.
- **Framework Reuse.** The use of collections of classes that together implement the basic functionality of a common technical or business domain. Horizontal frameworks, such as a security framework or user interface framework such as the Java Foundation Class (JFC) library and vertical frameworks, such as a financial services framework, are common.
- **Artifact Reuse.** The use of previously created development artifacts – use cases, standards documents, domain-specific models, procedures and guidelines, and even other applications such as a commercial off the shelf (COTS) package – to give you a kick start on a new project. Sometimes you take the artifact as is and other times you simply use it as an example to give you an idea about how to proceed.
- **Module Reuse.** The use of pre-built, fully encapsulated "modules" – components, services, or code libraries – in the

development of your application. A module is self sufficient and encapsulates only one concept. Module reuse differs from code reuse in that you don't have access to the source code.

- **Template Reuse.** The practice of using a common set of layouts for key development artifacts – documents, models, and source code – within your organization. I have some documentation templates available at the Agile Modeling (AM) Downloads page.
- **Code Reuse.** The reuse of source code within sections of an application and potentially across multiple applications. At its best code reuse is accomplished through the sharing of common classes and/or collections of functions and procedures. At its worst code reuse is accomplished by copying and then modifying existing code causing a maintenance nightmare.

You can address these reuse categories simultaneously. Framework reuse often locks you into the architecture of that framework, as well as the standards and guidelines used by the framework, but you can still take advantages of the other approaches to reuse in combination with framework reuse. Artifact and module reuse are the easiest places to start, with a little bit of research you can find reusable items quickly. However, if your organization doesn't have a well-defined development process that it follows you may get little benefit from templates. Pattern reuse is typically the domain of developers with good modeling skills and your enterprise architects should be publishing and providing pattern-oriented guidance to them.

It is important to note that although Figure 1 indicates that pattern reuse is generally more effective than artifact reuse you may discover that within your organization the opposite holds true. This may occur because you have a comprehensive collection of reusable artifacts in place, because your organization culture is conducive to artifact reuse, or simply because your developers have little experience with patterns.

---

## 14.2 INTRODUCTION

---

Object-oriented programming is often touted as promoting software reuse. Languages like Smalltalk are claimed to reduce not only development time but also the cost of maintenance, simplifying the creation of new systems and of new versions of old systems. This is true, but object-oriented programming is not a panacea. Program components must be designed for reusability. There is a set of design techniques that makes object-oriented software more reusable. Many of these techniques are widely used within the object-oriented programming community, but few of them have ever been written down. This article describes and organizes these techniques. It uses Smalltalk vocabulary, but most of what it says applies to other object-oriented languages. It concentrates on single inheritance and says little about multiple inheritance.

The first describes the attributes of object-oriented languages that promote reusable software. Data abstraction encourages modular systems that are easy to understand. Inheritance allows subclasses to share methods defined in superclasses, and permits programming-by-difference. Polymorphism makes it easier for a given component to work correctly in a wide range of new contexts. The combination of these

features makes the design of object-oriented systems quite different from that of conventional systems.

The middle section discusses frameworks, toolkits, and the software lifecycle. A framework is a set of classes that embodies an abstract design for solutions to a family of related problems, and supports reuses at a larger granularity than classes. During the early phases of a system's history, a framework makes heavier use of inheritance and the software engineer must know how a component is implemented in order to reuse it. As a framework becomes more refined, it leads to "black box" components that can be reused without knowing their implementations.

The last section gives a set of design rules for developing better, more reusable object-oriented programs. These rules can help the designer create standard protocols, abstract classes, and object-oriented frameworks.

As with any design task, designing reusable classes requires judgement, experience, and taste. However, this paper has organized many of the design techniques that are widely used within the object-oriented programming community so that new designers can acquire those skills more quickly.

---

## 14.3 OBJECT-ORIENTED PROGRAMMING

---

An object is similar to a value in an abstract data type---it encapsulates both data and operations on that data. Thus, object-oriented languages provide modularity and information-hiding, like other modern languages. Too much is made of the similarities of data abstraction languages and object-oriented languages. In our opinion, all modern languages should provide data abstraction facilities. It is therefore more important to see how object-oriented languages differ from conventional data abstraction languages.

There are two features that distinguish an object-oriented language from one based on abstract data types: polymorphism caused by late-binding of procedure calls and inheritance. Polymorphism leads to the idea of using the set of messages that an object understands as its type, and inheritance leads to the idea of an abstract class. Both are important.

### ***Polymorphism***

Operations are performed on objects by "sending them a message" (The object-oriented programming community does not have a standardized vocabulary. While "sending a message" is the most common term, and is used in the Smalltalk and Lisp communities, C++ programmers refer to this as "calling a virtual function".) Messages in a language like Smalltalk should not be confused with those in distributed operating systems. Smalltalk messages are just late-bound procedure calls. A message send is implemented by finding the correct method (procedure) in the class of the receiver (the object to which the message is sent), and invoking that method. Thus, the expression  $a + b$  will invoke different methods depending upon the class of the object in variable  $a$ .

Message sending causes polymorphism. For example, a method that sums the elements in an array will work correctly whenever all the elements of the array understand the addition message, no matter what classes they are in. In fact, if array elements are accessed by sending messages to the array, the procedure will work whenever it is given an argument that understands the array accessing messages.

Polymorphism is more powerful than the use of generic procedures and packages in Ada. A generic can be instantiated by macro substitution, and the resulting procedure or package is not at all polymorphic. On the other hand, a Smalltalk object can access an array in which each element is of a different class. As long as all the elements understand the same set of messages, the object can interact with the elements of the array without regard to their class. This is particularly useful in windowing systems, where the array could hold a list of windows to be displayed. This could be simulated in Ada using variant records and explicitly checking the tag of each window before displaying it, thus ensuring that the correct display procedure was called. However, this kind of programming is dangerous, because it is easy to forget a case. It leads to software that is hard to reuse, since minor modifications are likely to add more cases. Since the tag checks will be widely distributed through the program, adding a case will require wide-spread modifications before the program can be reused.

### ***Protocol***

The specification of an object is given by its *protocol*, i.e. the set of messages that can be sent to it. The type of the arguments of each message is also important, but "type" should be thought of as protocol and not as class. For a discussion of types in Smalltalk, see [Johnson 1986]. Objects with identical protocol are interchangeable. Thus, the interface between objects is defined by the protocols that they expect each other to understand. If several classes define the same protocol then objects in those classes are "plug compatible". Complex objects can be created by interconnecting objects from a set of compatible components. This gives rise to a style of programming called *building tool kits*, of which more will be said later.

Although protocols are important for defining interfaces within programs, they are even more important as a way for programmers to communicate with other. Shared protocols create a shared vocabulary that programmers can reuse to ease the learning of new classes. Just as mathematicians reuse the names of arithmetic operations for matrices, polynomials, and other algebraic objects, so Smalltalk programmers use the same names for operations on many kinds of classes. Thus, a programmer will know the meaning of many of the components of a new program the first time it is read.

Standard protocols are given their power by polymorphism. Languages with no polymorphism at all, like Pascal, discourage giving different procedures the same name, since they then cannot be used in the same program. Thus, many Pascal programs use a large number of slightly different names, such as *MatrixPlus*, *ComplexPlus*, *PolynomialPlus*, etc. Languages that use generics and overloading to provide a limited form of polymorphism can benefit from the use of standard protocols, but the benefits do not seem large enough to have forced wide use of them. (Booch shows how standard protocols might be

used in Ada. In Smalltalk, however, there are a wide number of well-known standard protocols, and all experienced programmers use them heavily.

Standard protocols form an important part of the Smalltalk culture. A new programmer finds it much easier to read Smalltalk programs once standard protocols are learned, and they form a standard vocabulary that ensures that new components will be compatible with old.

### ***Inheritance***

Most object-oriented programming languages have another feature that differentiates them from other data abstraction languages; class inheritance. Each class has a superclass from which it inherits operations and internal structure. A class can add to the operations it inherits or can redefine inherited operations. However, classes cannot delete inherited operations.

Class inheritance has a number of advantages. One is that it promotes code reuse, since code shared by several classes can be placed in their common superclass, and new classes can start off having code available by being given a superclass with that code. Class inheritance supports a style of programming called *programming-by-difference*, where the programmer defines a new class by picking a closely related class as its superclass and describing the differences between the old and new classes. Class inheritance also provides a way to organize and classify classes, since classes with the same superclass are usually closely related.

One of the important benefits of class inheritance is that it encourages the development of the standard protocols that were earlier described as making polymorphism so useful. All the subclasses of a particular class inherit its operations, so they all share its protocol. Thus, when a programmer uses programming-by-difference to rapidly build classes, a family of classes with a standard protocol results automatically. Thus, class inheritance not only supports software reuse by programming-by-difference, it also helps develop standard protocols.

Another benefit of class inheritance is that it allows extensions to be made to a class while leaving the original code intact. Thus, changes made by one programmer are less likely to affect another. The code in the subclass defines the differences between the classes, acting as a history of the editing operations.

Not all object-oriented programming languages allow protocol and inheritance to be separated. Languages like C++ [Stroustrup 1986] that use classes as types require that an object have the right superclass to receive a message, not just that it have the right protocol. Of course, languages with multiple inheritance can solve this problem by associating a superclass with every protocol.

---

## **14.4 ABSTRACT CLASSES**

---

Standard protocols are often represented by *abstract classes*.

An abstract class never has instances, only its subclasses have instances. The roots of class hierarchies are usually abstract classes,

while the leaf classes are never abstract. Abstract classes usually do not define any instance variables. However, they define methods in terms of a few undefined methods that must be implemented by the subclasses. For example, class `Collection` is abstract, and defines a number of methods, including `select:`, `collect:`, and `inject:into:`, in terms of an iteration method, `do:`. Subclasses of `Collection`, such as `Array`, `Set`, and `Dictionary`, define `do:` and are then able to use the methods that they inherited from `Collection`. Thus, abstract classes can be used much like program skeletons, where the user fills in certain options and reuses the code in the skeleton.

A class that is not abstract is *concrete*. In general, it is better to inherit from an abstract class than from a concrete class. A concrete class must provide a definition for its data representation, and some subclasses will need a different representation. Since an abstract class does not have to provide a data representation, future subclasses can use any representation without fear of conflicting with the one that they inherited.

Creating new abstract classes is very important, but is not easy. It is always easier to reuse a nicely packaged abstraction than to invent it. However, the process of programming in Smalltalk makes it easier to discover the important abstractions. A Smalltalk programmer always tries to create new classes by making them be subclasses of existing ones, since this is less work than creating a class from scratch. This often results in a class hierarchy whose top-most class is concrete. The top of a large class hierarchy should almost always be an abstract class, so the experienced programmer will then try to reorganize the class hierarchy and find the abstract class hidden in the concrete class. The result will be a new abstract class that can be reused many times in the future.

An example of a Smalltalk class that needs to be reorganized is `View`, which defines a user-interface object that controls a region of the screen. `View` has 27 subclasses in the standard image, but is concrete. A careful examination reveals a number of assumptions made in `View` that most of its subclasses do not use. The most important is that each view will have subviews. In fact, most subclasses of `View` implement views that can never have subviews. Quite a bit of code in `View` deals with adding and positioning subviews, making it very difficult for the beginning programmer to understand the key abstractions that `View` represents. The solution is simple: split `View` into two classes, one (`View`) of which is the abstract superclass and the other (`ViewWithSubviews`) of which is a concrete subclass that implements the ability to have subviews. The result is much easier to understand and to reuse.

### ***Inheritance vs. decomposition***

Since inheritance is so powerful, it is often overused. Frequently a class is made a subclass of another when it should have had an instance variable of that class as a component. For example, some object-oriented user-interface systems make windows be a subclass of `Rectangle`, since they are rectangular in shape. However, it makes more sense to make the rectangle be an instance variable of the window. Windows are not necessarily rectangular, rectangles are better thought



of as geometric values whose state cannot be changed, and operations like moving make more sense on a window than on a rectangle.

Behavior can be easier to reuse as a component than by inheriting it. There are at least two good examples of this in Smalltalk-80. The first is that a parser inherits the behavior of the lexical analyzer instead of having it as a component. This caused problems when we wanted to place a filter between the lexical analyzer and the parser without changing the standard compiler. The second example is that scrolling is an inherited characteristic, so it is difficult to convert a class with vertical scrolling into one with no scrolling or with both horizontal and vertical scrolling. While multiple inheritance might solve this problem, it has problems of its own. Moreover, this problem is easy to solve by making scrollbars be components of objects that need to be scrolled.

Most object-oriented applications have many kinds of hierarchies. In addition to class inheritance hierarchies, they usually have *instance hierarchies* made up of regular objects. For example, a user-interface in Smalltalk consists of a tree of views, with each subview being a child of its superview. Each component is an instance of a subclass of View, but the root of the tree of views is an instance of StandardSystemView. As another example, the Smalltalk compiler produces parse trees that are hierarchies of parse nodes. Although each node is an instance of a subclass of ParseNode, the root of the parse tree is an instance of MethodNode, which is a particular subclass. Thus, while View and ParseNode are the abstract classes at the top of the class hierarchy, the objects at the top of the instance hierarchy are instances of StandardSystemView and MethodNode.

This distinction seems to confuse many new Smalltalk programmers. There is often a phase when a student tries to make the class of the node at the top of the instance hierarchy be at the top of the class hierarchy. Once the disease is diagnosed, it can be easily cured by explaining the differences between the instance and class hierarchies.

---

## 14.5 SOFTWARE REUSE

---

One of the reasons that object-oriented programming is becoming more popular is that software reuse is becoming more important. Developing new systems is expensive, and maintaining them is even more expensive. A recent study by Wilma Osborne of the National Bureau of Standards suggests that 60 to 85 percent of the total cost of software is due to maintenance. Clearly, one way to reuse a program is to enhance it, so maintenance is a special case of software reuse. Both require programmers to understand and modify software written by others. Both are difficult.

Evolutionary lifecycles are the rule rather than the exception. Software maintenance can be categorized as corrective, adaptive, and perfective. Corrective maintenance is the process of diagnosing and correcting errors. Adaptive maintenance consists of those activities that are needed to properly integrate a software product with new hardware, peripherals, etc. Perfective maintenance is required when a software product is successful. As such a product is used, pressure is brought to bear on the developers to enhance and extend the functionality of that product. Osborne reports that perfective maintenance accounts for 60

percent of all maintenance, while adaptive and corrective maintenance each account for about 20 percent of maintenance. Since 60% of maintenance activity is perfective, an evolutionary phase is an important part of the lifecycle of a successful software product.

We have already seen that object-oriented programming languages encourage software reuse in a number of ways. Class definitions provide modularity and information hiding. Late-binding of procedure calls means that objects require less information about each other, so objects need only to have the right protocol. A polymorphic procedure is easier to reuse than one that is not polymorphic, because it will work with a wider range of arguments. Class inheritance permits a class to be reused in a modified form by making subclasses from it. Class inheritance also helps form the families of standard protocols that are so important for reuse.

These features are also useful during maintenance. Modularity makes it easier to understand the effect of changes to a program. Polymorphism reduces the number of procedures, and thus the size of the program that has to be understood by the maintainer. Class inheritance permits a new version of a program to be built without affecting the old.

Many of the techniques for reusing software written in conventional languages are paralleled by object-oriented techniques. For example, program skeletons are entirely subsumed by abstract classes. Copying and editing a program is subsumed by inheriting a class and overriding some of its methods. The object-oriented techniques have the advantage of giving the new class only the differences between it and the old, making it easier to determine how a new program differs from the old. Thus, a set of subclasses preserves the history of changes made to the superclass by its subclasses. Conditionalizing a program by adding flag parameters or variant tag tests can almost always be replaced by making a subclass for each variant and having the subclasses override the methods making the tests.

Software reuse does not happen by accident, even with object-oriented programming languages. System designers must plan to reuse old components and must look for new reusable components. The Smalltalk community practices reuse very successfully. The keys to successful software reuse are attitude, tools, and techniques.

Smalltalk programmers have a different attitude than other programmers. There is no shame in borrowing system classes or classes invented by other programmers. Rewriting an old class to make it easier to reuse is as important as inventing a new class. A new class that is not compatible with old classes is looked down upon. Smalltalk programmers expect to spend as much time reading old code to see how to reuse it as writing new code. In fact, writing a Smalltalk program is very similar to maintaining programs written in other languages, in that it is just as important for the new software to fit in as it is for it to be efficient and easy to understand.

The most important attitude is the importance given to the creation of reusable abstractions. Kent Beck describes the difficulty in

finding reusable abstractions and the importance placed on them by saying:

Even our researchers who use Smalltalk every day do not often come up with generally useful abstractions from the code they use to solve problems. Useful abstractions are usually created by programmers with an obsession for simplicity, who are willing to rewrite code several times to produce easy-to-understand and easy-to-specialize classes.

---

## 14.6 TOOLKITS AND FRAMEWORKS

---

One of the most important kinds of reuse is reuse of designs. A collection of abstract classes can be used to express an abstract design. The design of a program is usually described in terms of the program's components and the way they interact. For example, a compiler can be described as consisting of a lexer, a parser, a symbol table, a type checker, and a code generator.

An object-oriented abstract design, also called a *framework*, consists of an abstract class for each major component. (Apparently the name for frameworks at Xerox Information Systems is "teams".) The interfaces between the components of the design are defined in terms of sets of messages. There will usually be a library of subclasses that can be used as components in the design. A compiler framework would probably have some concrete symbol table classes and some classes that generate code for common machines. In theory, code generators could be mixed with many different parsers. However, parsers and lexers would be closely matched. Thus, some parts of a framework place more constraints on each other than others.

MacApp is a framework for Macintosh applications. An abstract MacApp application consists of one or more windows, one or more documents, and an application object. A window contains a set of views, each of which displays part of the state of a document. MacApp also contains commands, which automate the undo/redo mechanism, and printer handlers, which provide device independent printing. Most application classes do little besides define the class of their document. They inherit a command interpreter and menu options. Most document classes do little besides define their window and how to read and write documents to disk. They inherit menu options for saving the documents and tools for selecting which document to open next. An average programmer rarely makes new window classes, but usually has to define a view class that renders an image of a document. MacApp not only ensures that programs meet the Macintosh user-interface standard, but makes it much easier to write interactive programs.

Other frameworks include the Lisa Toolkit, which was used to build applications for the Lisa desktop environment, and the Smalltalk Model/View/Controller (MVC), which is a framework for constructing Smalltalk-80 user interfaces. Although these frameworks are concerned primarily with implementing a standard user interface, frameworks are by no means limited to the user interface. For example, the Battery Simulation is a framework for constructing realtime psychophysiological experiments.

Frameworks are useful for reusing more than just mainline application code. They can also describe the abstract designs of library

components. The ability of frameworks to allow the extension of existing library components is one of their principal strengths.

Frameworks are more than well written class libraries. A good example of a set of library utility class definitions is the Smalltalk Collection hierarchy. These classes provide ways of manipulating collections of objects such as Arrays, Dictionaries, Sets, Bags, and the like. In a sense, these tools correspond to the sorts of tools one might find in the support library for a conventional programming system. Each component in such a library can serve as a discrete, stand-alone, context independent part of a solution to a large range of different problems. Such components are largely application independent.

A framework, on the other hand, is an abstract design for a particular kind of application, and usually consists of a number of classes. These classes can be taken from a class library, or can be application-specific.

Frameworks can be built on top of other frameworks by sharing abstract classes. FOIBLE is a framework for building "device programming" systems in Smalltalk. It lets the user edit a picture consisting of a collection of interconnected devices. These devices have computational meaning, so editing the picture is a form of programming. FOIBLE uses the MVC framework to implement the editor, but adds Tools and Foibles to implement the semantics of the picture and the visual representation of components. Thus, FOIBLE is built on top of MVC.

Frameworks provide a way of reusing code that is resistant to more conventional reuse attempts. Application independent components can be reused rather easily, but reusing the edifice that ties the components together is usually possible only by copying and editing it. Unlike skeleton programs, which is the conventional approach to reusing this kind of code, frameworks make it easy to ensure the consistency of all components under changing requirements.

Since frameworks provide for reuse at the largest granularity, it is no surprise that a good framework is more difficult to design than a good abstract class. Frameworks tend to be application specific, to interlock with other frameworks by sharing abstract classes, and to contain some abstract classes that are specialized for the framework. Designing a framework requires a great deal of experience and experimentation, just like designing its component abstract classes.

---

## **14.7 WHITE-BOX VS. BLACK-BOX FRAMEWORKS**

---

One important characteristic of a framework is that the methods defined by the user to tailor the framework will often be called from within the framework itself, rather than from the user's application code. The framework often plays the role of the main program in coordinating and sequencing application activity. This inversion of control gives frameworks the power to serve as extensible skeletons. The methods supplied by the user tailor the generic algorithms defined in the framework for a particular application.

A framework's application specific behavior is usually defined by adding methods to subclasses of one or more of its classes. Each method added to a subclass must abide by the internal conventions of its superclasses. We call these *white-box* frameworks because their implementation must be understood to use them.

A good example is the MVC Controller class, which maps user actions into messages to the application. When the mouse moves into the region of a controller, it is sent the `startUp` message, which causes the controller to be sent the `controlInitialize`, `controlLoop`, and `controlTerminate` messages, in that order. The behavior of a controller when it is selected and deselected is changed by redefining `controlInitialize` and `controlTerminate`. The default behavior of `controlLoop` is to repeatedly send the controller the `controlActivity` message until the mouse moves out of the region of the controller. Thus, the reaction of a controller to mouse movement, mouse button clicks, and keyboard events is determined by the definition of the `controlActivity`.

The major problem with such a framework is that every application requires the creation of many new subclasses. While most of these new subclasses are simple, their number can make it difficult for a new programmer to learn the design of an application well enough to change it.

A second problem is that a white-box framework can be difficult to learn to use, since learning to use it is the same as learning how it is constructed.

Another way to customize a framework is to supply it with a set of components that provide the application specific behavior. Each of these components will be required to understand a particular protocol. All or most of the components might be provided by a component library. The interface between components can be defined by protocol, so the user needs to understand only the external interface of the components. Thus, this kind of a framework is called a *black-box* framework.

There is a set of black-box components of MVC called the *pluggable views*. These components were designed with the realization that the majority of MVC classes that were created were controllers with a customized menu. The pluggable views let controllers take the menus as parameters, thus greatly reducing the need to create new controller classes. Most of the programming tools in the latest versions of Smalltalk-80, such as the browser, file tool, and debugger, use pluggable views and do not require any new user interface classes. The method that invokes a tool will create instances of the various components, send messages to them to customize them for the tool, and connect them together.

Black-box frameworks like the pluggable views are easier to learn to use than white-box frameworks, but are less flexible. Pluggable views are usually sufficient to describe user interfaces that display only text, but the user who wants a more graphical user interface will have to use the original MVC framework. Fortunately, pluggable views fit into the MVC framework well, so the user only has to create components to handle the graphical aspects of the interface.

One way of characterizing the difference between white-box and black-box frameworks is to observe that in white-box frameworks, the state of each instance is implicitly available to all the methods in the framework, much as the global variables in a Pascal program are. In a black-box framework, any information passed to constituents of the framework must be passed explicitly. Hence, a white-box framework relies on the intra-object scope rules to allow it to evolve without forcing it to subscribe to an explicit, rigid protocol that might constrain the design process prematurely.

A framework becomes more reusable as the relationship between its parts is defined in terms of a protocol, instead of using inheritance. In fact, as the design of a system becomes better understood, black-box relationships should replace white-box ones. Black-box relationships are an ideal towards which a system should evolve.

---

## 14.8 TOOLKITS

---

An object-oriented application construction environment, or toolkit, is a collection of high level tools that allow a user to interact with an application framework to configure and construct new applications. Examples of toolkits are Alexander's Glazier system for constructing Smalltalk-80 MVC applications, and Smith's Alternate Reality Kit. All toolkits are based on one or more frameworks.

One of the advantages of black-box frameworks is that they are better at serving as the foundation of a toolkit. It is easy to build a tool that lets a user choose prebuilt components and connect them together, and a successful black-box framework permits most applications to be constructed that way. An example of such a tool is Glazier, which builds an application within the Model/View/Controller framework using pluggable views.

Frameworks make it easier to define specialized programs for constructing classes. For example, a compiler might provide tools for building parsers, lexers, and code generators. It is easier to build a tool for constructing classes with well defined interfaces than it is to build a general purpose automatic programming system.

---

## 14.9 LIFECYCLE

---

The lifecycle of a Smalltalk application is not necessarily different from that of other programs developed using rapid prototyping. However, the lifecycle of classes differs markedly from that of program components in conventional languages, since classes may be reused in many applications.

Classes usually start out being application dependent. It is always worthwhile to examine a nearly-complete project to see if new abstract classes and frameworks can be discovered. They can probably be reused in later projects, and their presence in the current project will make later enhancements much easier. Thus, creating abstract classes and frameworks is both a way of scavenging components for later reuse and a way of cleaning up a design. The final class hierarchy is a

description of how the system ought to have been designed, though it may bear little relation to the original design.

One of the reasons that Smalltalk is good for prototyping is that the programmer can borrow code from anywhere in the system. However, this should never be mistaken for good design. It is almost always necessary at the end of a project to reorganize the class hierarchy. Unfortunately, few tools help this task. The rules section will discuss how to recognize class hierarchies that need to be reorganized. Suggestions for tools to aid reorganization will appear in the tools section.

One sign that a good abstraction has been found is that code size decreases, indicating that code is being reused. Many Smalltalk projects have periods in which the size of the code increases at a steady rate, followed by periods in which little change occurs to the code, followed by a sharp decrease in the size of the code. Code size increases as the programmers add new classes and new methods to old classes. Eventually the programmers realize that they need to rearrange the class hierarchy. They spend a bit of time in debate and experimentation and then make the necessary changes, usually creating a new abstract class or two. Since Smalltalk programs tend to be compact, it is feasible to rewrite a system many times during its development. The result is much easier to understand and maintain than typical nonobject-oriented systems.

There are many ways that classes can be reorganized. Big, complex classes can be split into several smaller classes. A common superclass can be found for a set of related classes. Concrete superclasses can be made abstract. A white-box framework can be converted into a black-box framework. All these changes make classes more reusable and maintainable.

Every class hierarchy offers the possibility of becoming a framework. Since a white-box framework is just a set of conventions for overriding methods, there is no fine line between a white-box framework and a simple class hierarchy. In its simplest form, a white-box framework is a program skeleton, and the subclasses are the additions to the skeleton.

Ideally, each framework will evolve into a black-box framework. However, it is often hard to tell in advance how a white-box framework will evolve into a black-box framework, and many frameworks will not complete the journey from skeleton to black-box frameworks during their lifetimes.

White-box inheritance frameworks should be seen as a natural stage in the evolution of a system. Because they are a middle ground between a particular application and an abstract design, white-box inheritance frameworks provide an indispensable path along which applications may evolve. A white-box framework will sometime be a waystation in the evolution of a loose collection of methods into a discrete set of components. At other times, a white-box framework will be a finished product. A useful design strategy is to begin with a white-box approach. White-box frameworks, as a result of their internal informality, are usually relatively easy to design. As the system evolves, the designer can then see if additional internal structure emerges.



## REUSE : LIBRARIES, FRAME WORKS COMPONENTS AND PATTERNS-II

### Unit Structure

- 15.1 Design methodology
- 15.2 Rules for Finding Standard Protocols
- 15.3 Rules for Finding Abstract Classes
- 15.4 Object-Oriented Programming Tools
- 15.5 Conclusion

---

### 15.1 DESIGN METHODOLOGY

---

The product of an object-oriented design is a list of class definitions. Each class has a list of operations that it defines and a list of objects with which its instances communicate. In addition, each operation has a list of other operations that it will invoke. A design is complete when every object that is referenced has been defined and every operation is defined. The design process incrementally extends an incomplete design until it is complete.

Object-oriented design starts with objects. Booch suggests that the designer start with a natural language description of the desired system and use the nouns as a starting point for the classes of objects to be designed. Each verb is an operation, either one implemented by a class or one used by the class. The resulting list of classes and operations can be used as the start of the design process.

Operations on an object are always thought about from the object's point of view. Thus, instead of displaying an object, an object is asked to display itself. Methods are receiver-centric---many of the comments in the standard Smalltalk-80 image use the word "I" to refer to the receiver. This is in stark contrast to other ways of programming, where "The use of anthropomorphic terminology when dealing with computer systems is a sign of professional immaturity".

Booch's design methodology defines classes for objects in the problem domain. However, classes are often needed for operations in the problem domain. For example, compiling a program can be thought of as an operation on programs. However, because compilation is so complex, it is best to have separate compiler objects to represent compilation. The *compile* operation on programs would make a new compiler object and use it to make a compiled version of the program.

It can be difficult to decide whether an operation should be implemented as a method in a class or as a separate class. Halbert and O'Brien discuss this problem at length. In general, there is no absolute



way to decide, but positive answers to the following questions all indicate that a new class should be created.

1. Is the operation a meaningful abstraction?
2. Is the operation likely to be shared by several classes?
3. Is the operation complex?
4. Does the operation make little use of the representation of its operands?
5. Will relatively few users of the class want to use the operation?

The compiler example shows that it is possible to make an operation both a class and a method by having the method make an object of the class. This separates the implementation of the operation from that of the class and makes it more reusable, but permits the user to continue to think of the operation as a method.

It can also be difficult to decide which class should implement an operation. Operations with several arguments can frequently be implemented as methods in the classes of any of its arguments. The rules listed above can also be used to make this decision. For example, if an operation does not send messages to an object or access its instance variables then it should not be in the object's class.

We are not implying that classes can be reorganized mechanically. A class should represent a well-defined abstraction, not just a bundle of methods and variable definitions. Human judgment is needed to decide when and how a class hierarchy is to be reorganized. Nevertheless, the following rules will frequently point out the need for a reorganization and suggest how it is to be accomplished.

---

## **15.2 RULES FOR FINDING STANDARD PROTOCOLS**

---

It is very important that the design process result in standard protocols. In other words, many of the classes should have nearly identical external interfaces and there should be sets of operations that many classes implement.

Standard protocols are developed by choosing names carefully. The need for standard protocols is one reason why it takes a long time to become an expert Smalltalk programmer. Many of the more important protocols are described in the Blue Book [Goldberg & Robson 1983], but just as many are not documented anywhere except in the source code. Thus, the only way to learn these protocols is by experience.

There are a number of rules of thumb that will help develop standard protocols. A programmer practicing these rules is more likely to keep from giving different names to the same operation in different classes. These rules help minimize the number of different names and maximize the number of names shared by a set of classes.

**Rule 1: *Recursion introduction.***

If one class communicates with a number of other classes, its interface to each of them should be the same. If an operation X is implemented by performing a similar operation on the components of the receiver, then that operation should also be named X. Even if the name of the operation has to be changed to add more arguments, (Smalltalk message names indicate the number of arguments to the message.) it makes sense to make the names similar so that readers of the program will note the connection. The result is that a method for a message sends that same message to other objects. If the other objects are in the same class as the sender then the method is recursive. Even if no real recursion exists, the method appears recursive, so we call this rule *recursion introduction*.

Recursion introduction can help decide the class in which an operation should be a method. Consider the problem of converting a parse tree into machine language. In addition to an object representing the parse tree, there will be an object representing the final machine language procedure. The ``generate code" message could be sent to either object. However, the best design is to implement the generate code message in the parse tree class, since a parse tree will consist of many parse nodes, and a parse node will generate machine code for itself by recursively asking its subtrees to generate code for themselves.

**Rule 2: *Eliminate case analysis.***

It is almost always a mistake to explicitly check the class of an object. Code of the form

```
anObject class == ThisClass ifTrue: [anObject foo] ifFalse: [anObject
fee]
```

should be replaced with a message to the object whose class is being checked. Methods will have to be created in the various possible classes of the object to respond to the message, and each method will contain one of the cases that is being replaced.

Case analysis of the values of variables is usually a bad idea, too. For example, a parse tree might contain nodes that represent instance variables, global variables, method arguments, and temporary variables. The Smalltalk-80 compiler uses one class to represent all these kinds of variables and differentiates between them on the value of an instance variable. It would be better to have a separate class for each kind of variable.

Eliminating case analysis is more difficult when the cases are accessing instance variables, but it is no less important. If instance variables are being accessed then *self* will need to be an argument to the message and more messages may need to be defined to access the instance variables.

**Rule 3: *Reduce the number of arguments.***

Messages with half a dozen or more arguments are hard to read. Except for instance creation messages, a message with this many arguments should be redefined. When a message has a smaller number of arguments it is more likely to be similar to some other message, thus increasing the possibility of giving them the same name.

The number of arguments can be reduced by breaking a message into several smaller messages or by creating a new class that represents a group of arguments. Frequently there will be several kinds of messages that pass the same set of objects around. This set of objects is essentially a new object, and the design can be changed to reflect that fact by replacing the set of objects with an object that contains them.

**Rule 4: *Reduce the size of methods.***

Well-designed Smalltalk methods are almost always small. It is easier to subclass a class with small methods, since its behavior can be changed by redefining a few small methods instead of modifying a few large methods. A thirty line method is large and probably needs to be broken into pieces. Often a method in a superclass is split when a subclass is made. Most of the inherited method is correct, but one part needs to be changed. Instead of rewriting the entire method, it is split into pieces and the one piece that has changed is redefined. This change leaves the superclass even easier to subclass.

These design rules are all related, since eliminating cases reduces the size of methods, breaking a method into pieces is likely to reduce the number of arguments that any one method needs, and reducing the number of arguments is likely to create more methods with the same name.

---

**15.3 RULES FOR FINDING ABSTRACT CLASSES**

---

**Rule 5: *Class hierarchies should be deep and narrow.***

A well developed class hierarchy should be several layers deep. A class hierarchy consisting of one superclass and 27 subclasses is much too shallow. A shallow class hierarchy is evidence that change is needed, but does not give any idea how to make that change.

An obvious way to make a new superclass is to find some sibling classes that implement the same message and try to migrate the method to a common superclass. Of course, the classes are likely to provide different methods for the message, but it is often possible to break a method into pieces and place some of the pieces in the superclass and some in the subclasses. For example, displaying a view consists of displaying its border, displaying its subviews, and displaying its contents. The last part must be implemented by each subclass, but the others are inherited from View.

**Rule 6: *The top of the class hierarchy should be abstract.***

Inheritance for generalization or code sharing usually indicates the need for a new subclass. If class B overrides a method *x* that it inherits from class A then it might be better to move the methods in A that B does inherit to C, a new superclass of A, as shown in Figure 1. C will probably be abstract. B can then become a subclass of C, and will not have to redefine any methods. Instance variables or methods defined in A that are used by B should be moved to C.

**Rule 7: *Minimize accesses to variables.***

Since one of the main differences between abstract and concrete classes is the presence of data representation, classes can be made more abstract by eliminating their dependence on their data representation. One way this can be done is to access all variables by sending messages. The data representation can be changed by redefining the accessing messages.

**Rule 8: *Subclasses should be specializations.***

There are several different ways that inheritance can be used. *Specialization* is the ideal that is usually described, where the elements of the subclass can all be thought of as elements of the superclass. Usually the subclass will not redefine any of the inherited methods, but will add new methods. For example, a two dimensional array is a subclass of Array in which all the elements are arrays. It might have new messages that use two indexes, instead of just one.

An important special case of specialization is making *concrete classes*. Since an abstract class is not executable, making a subclass of an abstract class is different from making a subclass of a concrete class. The abstract class requires its subclasses to define certain operations, so making a concrete class is similar to filling in the blanks in a program template. An abstract class may define some operations in an overly general fashion, and the subclass may have to redefine them. For example, the *size* operation in class Collection is implemented by iterating over the collection and counting its elements. Most subclasses of Collection have an instance variable that contains the size, so *size* is redefined in those subclasses to return that instance variable.

***Rules for Finding Frameworks***

Large classes are frequently broken into several small classes as they grow, leading to a new framework. A collection of small classes can be easier to learn and will almost always be easier to reuse than a single large class. A collection of class hierarchies provides the ability to mix and match components while a single class hierarchy does not. Thus, breaking a compiler into a parsing phase and a code generation phase permits a new language to be implemented by building only a new parser, and a new machine to be supported by building only a new code generator.

**Rule 9 : *Split large classes.***

A class is supposed to represent an abstraction. If a class has 50 to 100 methods then it must represent a complicated abstraction. It is likely that such a class is not well defined and probably consists of several different abstractions. Large classes should be viewed with suspicion and held to be guilty of poor design until proven innocent.

**Rule10 : *Factor implementation differences into subcomponents.***

If some subclasses implement a method one way and others implement it another way then the implementation of that method is independent of the superclass. It is likely that it is not an integral part of the subclasses and should be split off into the class of a component.

Multiple inheritance can also be used to solve this problem. However, if an algorithm or set of methods is independent of the rest of the class then it is cleaner to encapsulate it in a separate component.

**Rule 11 : *Separate methods that do not communicate.***

A class should almost always be split when half of its methods access half of its instance variables and the other half of its methods access the other half of its variables. This sometimes occurs when there are several different ways to view objects in the class

For example, a complex graphical object may cache its image as a bitmap, but the image is derived from the complex structure of the object, which consists of a number of simple graphical objects. When the object is asked to display itself, it displays its cached image if it is valid. If the image is not valid, the object recalculates the image and displays it. However, the graphical object can also be considered a collection of (graphical) objects that can be added or removed. Changing the collection invalidates the image.

This graphical object could be implemented as a subclass of bitmapped images, or it could be a subclass of Collection. A system with multiple inheritance might make both be superclasses. However, it is best to make both the bitmap and the collection of graphical objects be components, since each of them could be implemented in a number of different ways, and none of those ways are critical to the implementation of the graphical object. Separating the bitmap class will make it easier to port the graphical object to a system with different graphics primitives, and separating the collection class will make it easier to make the graphical object be efficient even when very large.

**Rule 12: *Send messages to components instead of to self.***

An inheritance-based framework can be converted into a component-based framework black box structure by replacing overridden methods by message sends to components. Examples of such frameworks in conventional systems are sorting routines that take procedural parameters. Programs should be factored in this fashion whenever possible. Reducing the coupling between framework components so that the framework works with any plug-compatible object increases its cohesion and generality.

**Rule 13: *Reduce implicit parameter passing.***

Sometimes it is hard to split a class into two parts because methods that should go in different classes access the same instance variable. This can happen because the instance variable is being treated as a global variable when it should be passed as a parameter between methods. Changing the methods to explicitly pass the parameter will make it easier to split the class later.

---

**15.4 OBJECT-ORIENTED PROGRAMMING TOOLS**


---

There are at least two new kinds of tools needed for the style of programming we have just described. One kind helps the programmer reorganize class hierarchies and the other helps the programmer build applications from frameworks. Other tools would also be helpful, such as tools to help a programmer find components in libraries, but these will not be discussed.

It takes a great deal of inspiration to construct a good class hierarchy. However, it is possible to build tools that would let a programmer know that a class hierarchy had problems. These tools would be like the English style tools in the Unix writer's workbench. They would complain about perceived problems but would let the programmer decide whether the complaints were valid and how to fix them. Other tools could help reorganize the class hierarchy once a problem was diagnosed. For example, if a method of a superclass is ignored by its subclass then some abstractions in the superclass are not being inherited by the subclass. This is probably a case of subclassing for generalization or code sharing. It might be best to break the superclass into two classes, one a subclass of the other. The new subclass would have all the methods that are unused by the old subclasses. Similarly, a sign of inheritance for code sharing is that many of a superclass's methods are redefined. Perhaps some of these redefined methods should be in a concrete subclass, making the superclass abstract.

Reorganizing a class hierarchy is not difficult in Smalltalk, since it is easy to change the superclass of a class and to add and remove instance variables. It is difficult to copy a class, but copying is rarely needed. A more important problem is that the lack of type checking in Smalltalk means that if a method inherited by two subclasses is moved into one of the subclasses then no warning will be given until runtime. It is virtually impossible to reorganize class hierarchies without creating a few missing methods, though these are fortunately easy to fix. However, it is very difficult to change core class hierarchies, since any mistake will crash the system. It would be good to have tools for building reorganization plans and for inspecting the results of applying these plans. Consistency checks could help ensure that the plans would result in class hierarchies with the same behavior as the original ones.

The interfaces between components in a framework are not fixed but depend on the classes of the components. A particular component may place more restrictions on the other components. For example, if a scroll controller is used in a MVC triad then the view will have to be able to respond to scrolling messages. This requirement is evidenced by the fact that the controller will send some specialized messages to the view and so not every view will be compatible with it.

---

## 15.5 CONCLUSION

---

Smalltalk programmers often tell stories of how they built a complicated application in a few days. These experiences can occur only because the programmers are able to reuse so many software components and abstract designs. Building reusable components and designs takes much more time. However, it is time that pays off handsomely in the long run.

A number of factors account for the high reusability of object-oriented components. Polymorphism increases the likelihood that a given component will be usable in new contexts. Inheritance promotes the emergence of standard protocols, and allows existing components to be customized. Inheritance also promotes the emergence of abstract classes. Frameworks allow a collection of objects to serve as a template solution to a class of problems. Using frameworks, algorithms and control code, as well as individual components, can be reused.

