OBJECT-ORIENTED SOFTWARE ENGINEERING

UNIT I

Learning Objectives

- **Object Oriented Concepts-** Review of Object and Classes, Links and association, Generalization and specialization, Inheritance and Grouping concepts, Aggregation and Composition, Abstract Classes and Polymorphism, Metadata, Constraints, Reuse.
- **Object Oriented Methodologies-** Introduction to Rational Unified Process, Comparison of traditional life cycle models versus object oriented life cycle models.
- **UML-** Origin of UML, 4+1 view architecture of UML
- **Architecture-** Introduction, system development is model building, architecture, requirements model, analysis model,the design model, the implementation model, test model.

Evolution of Object Orientation

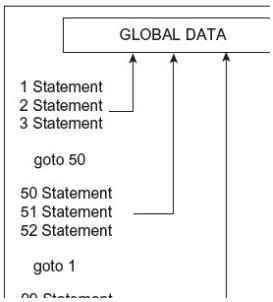
- The idea of object-oriented programming gained **momentum** in the 1970s and in the early 1980s.
- **Bjorn Stroustrup** integrated object-oriented programming into the C language. The resulting language was called C++ and it became the first object-oriented language to be widely used commercially.
- In the early 1990s a group at Sun led by **James Gosling** developed a simpler version of C++ called Java that was meant to be a programming language for video-on-demand applications.
- This project was going nowhere until the group **re-oriented** its focus and marketed Java as a language for programming Internet applications.
- The language has gained **widespread popularity** as the Internet has boomed, although its market penetration has been limited by its inefficiency.

Evolution of Object Orientation

1. **Monolithic Programming Approach:** In this approach, the program consists of **sequence of statements** that **modify data**.

- All the **statements** of the program are **Global** throughout the whole program. The **program control** is achieved through the use of **jumps** i.e. **goto statements**.
- In this approach, **code is duplicated** each time because there is no support for the function. **Data** is **not fully protected** as it can be accessed from any portion of the program.
- So this approach is useful for designing **small** and **simple** programs. The programming languages like ASSEMBLY and BASIC follow this approach.

Evolution of Object Orientation



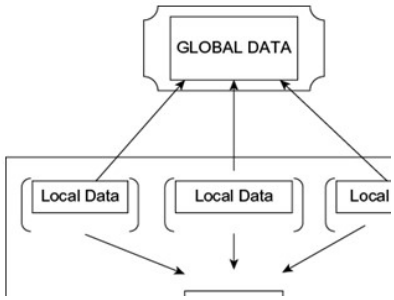
Evolution of Object Orientation

2. **Procedural Programming Approach:** This approach is **top down approach**. In this approach, a program is divided into **functions** that perform a **specific task**.

- This approach **avoids repetition of code** which is the main drawback of **Monolithic Approach**.
- The basic **drawback** of Procedural Programming Approach is that data is not secured because data is **global** and can be accessed by any function.
- This approach is mainly used for medium sized applications. The programming languages: **FORTRAN and COBOL** follow this approach.

•3. **Structured Programming Approach:** The basic principal of **structured programming approach** is to divide a program in functions and modules.

Evolution of Object Orientation



Evolution of Object Orientation

- The use of modules and functions makes the program more **comprehensible** (understandable). It helps to write **cleaner code** and helps to **maintain control** over each function. This approach gives importance to functions rather than data.
- It focuses on the development of large software applications. The programming languages: **PASCAL and C** follow this approach.

4. Object Oriented Programming Approach: The basic principal of the OOP approach is to **combine** both **data** and **functions** so that both can operate into a **single unit**. Such a unit is called an **Object**.

- This approach **secures data** also. Now a days this approach is used mostly in applications. The programming languages: **C++ and JAVA** follow this approach. Using this approach we can write any lengthy code.

Object Orientation Paradigm

- An approach to the solution of problems in which all **computations** are performed in context of objects.
- The objects are instances of **programming constructs**, normally called as **classes** which are **data abstractions** with **procedural abstractions** that operate on objects.
- A software system is a set of mechanism for performing certain **action** on certain data

  Algorithm + Data structure = Program

- **Data Abstraction + Procedural Abstraction**

Object Orientation

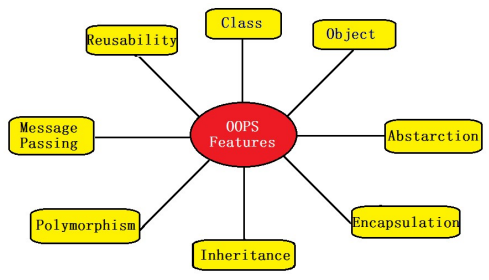
- **Object orientation** refers to a special type of programming paradigm that combines **data structures** with **functions** to create **re-usable objects.**
- The object-oriented (OO) paradigm is a **development strategy** based on the concept that systems should be **built** from a **collection of reusable components** called objects.
- Instead of separating data and functionality as is done in the structured paradigm, objects **encompass both**.
- **Why object orientation?**
  To create sets of **objects** that work together concurrently to produce s/w that better, model their problem domain that similarly system produced by traditional techniques.

Object Orientation Adaptation

Object orientation adapts to the following criteria's-

1. Changing requirements
2. Easier to maintain
3. More robust
4. Promote greater design
5. Code reuse
6. Higher level of abstraction
7. Encouragement of good programming techniques
8. Promotion of reusability

Object Orientated Features



Object Orientated Features

Object orientation adapts to the following criteria's-

1. Changing requirements
2. Easier to maintain
3. More robust
4. Promote greater design
5. Code reuse
6. Higher level of abstraction
7. Encouragement of good programming techniques
8. Promotion of reusability

Object Orientated Features

1. **OBJECT** - Object is a collection of number of **entities**. Objects take up space in the memory. Objects are **instances of classes**. When a program is executed , the objects interact by sending messages to one another. Each object contain **data** and **code** to manipulate the data. Objects can interact without having know details of each others data or code. **Each instance** of an object can hold its own **relevant data**.

2. **CLASS** - Class is a **collection** of **objects** of similar type. Objects are **variables** of the **type class**. Once a class has been defined, we can create any number of objects belonging to that class. Classes are **user define data types**. A class is a **blueprint** for any functional entity which defines its **properties** and its **functions**.

Object Orientated Features

3. **DATA ENCAPSULATION –** Combining data and functions into a single unit called **class** and the process is known as **Encapsulation**. Class variables are used for storing data and functions to specify various operations that can be performed on data. This process of **wrapping up** of data and functions that operate on data as a single unit is called as data encapsulation. Data is **not accessible** from the outside world and only those function which are present in the class can access the data.

4. **DATA ABSTRACTION**- Abstraction (from the Latinn *abs* means *away from* and *trahere* means to draw) is the **process** of taking away or **removing characteristics** from something in order to reduce it to a **set of essential characteristics.** Advantage of data abstraction is **security**.

Object Orientated Features

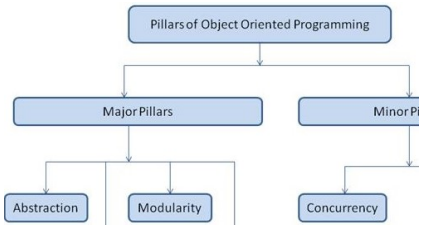
5. INHERITANCE- It is the process by which object of one class **acquire** the **properties** or features of objects of another class. The concept of inheritance provide the idea of reusability means we can add **additional features** to an existing class **without modifying it.** This is possible by driving a new class from the existing one. **Advantage** of inheritance is **reusability** of the **code**.

6. MESSAGE PASSING - The process by which **one object** can interact with **other object** is called **message passing**.

7. **POLYMORPHISM** - A greek term means **ability to take more than one form.** An operation may exhibit different behaviours in different instances. The behaviour depends upon the **types of data** used in the operation.

Object Orientated Features

8. PERSISTENCE - The process that allows the state of an **object** to be saved to **non-volatile storage** such as a file or a database and later restored even though the original creator of the object no longer exists.



Benefits of OOPs

• **Code Reuse and Recycling**:

Objects created for Object Oriented Programs can easily be reused in other programs.The code and designs in object-oriented software development are **reusable** because they are modeled directly out of the real-world problem-domain.

• **Design Benefits**:

Large programs are very difficult to write. Object Oriented Programs force designers to go through an extensive planning phase, which makes for **better designs** with **less flaws**.

• **Ease out development**: In addition, once a program reaches a certain size, Object Oriented Programs are actually *easier* to program than non-Object Oriented ones.

Benefits of OOPs

- **Object orientation works at a higher level of abstraction**

One of our most powerful techniques is the form of selective amnesia called 'Abstraction'. Abstraction allows us to ignore the details of a problem and concentrate on the whole picture.

- **Software life cycle requires no vaulting**

The object-oriented approach uses essentially the same language to talk about analysis, design, programming and (if using an Object-oriented DBMS) database design. This **streamlines** the entire software development process, reduces the level of **complexity** and **redundancy**, and makes for a cleaner system architecture and design.

Benefits of OOPs

- **Data is more stable than functions**

Functions are not the most stable part of a system, the data is. Over a period of time, the **requirements** of a system undergo radical change. New uses and needs for the software are discovered; new features are added and old features are removed. During the course of all this change, the **underlying heart- data** of the system remains comparatively constant.

- **Software Maintenance:**

Legacy code must be dealt with on a daily basis, either to be improved upon or made to work with newer computers and software. An Object Oriented Program is much easier to **modify** and **maintain**. So although a lot of work is spent before the program is written, less work is needed to maintain it over time.

Application Areas of OOPS

- Real time systems.
- Simulation & Modelling.
- Object-oriented database system.
- Object-oriented Operating System.
- Graphical User Interface.
- Window based O.S. design.
- Multimedia Design.
- CIM/CAD/CAM Systems.
- Computer based Training & Education System.
- AI and Expert System.
- Neural Networks and parallel programming.
- Decision support and office automation system.

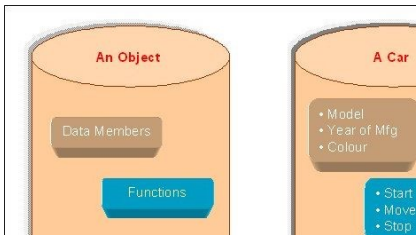
Object- *The CRUX of the matter!!*

- "An object is an **entity** which has a **state** and a defined set of **operations** which **operate** on that state."
- The **state** is represented as a set of object attributes. The operations associated with the object **provide services** to other objects (clients) which request these services when some computation is required
- Objects are **created** according to some **object class definition**. An object class definition serves as a **template** for objects. It includes **declarations** of all the attributes and services which should be associated with an object of that class.
- An Object is anything, **real** or **abstract**, about which we store data and those **methods** that manipulate the data.
- An **object** is a component of a program that knows how to perform certain actions and how to **interact** with other elements of the program.

Object- *The CRUX of the matter!!*

- Each **object** is an instance of a particular **class** or **subclass** with the class's own methods or procedures and data variables. An object is what **actually runs** in the computer.
- Objects are the basic run time entities in an **object oriented system**.
- They match closely with **real time objects**.
- Objects take up **space in memory** and have an associated **address** like a Record in Pascal and a Structure in C.
- Objects interact by **sending Message** to one other. E.g. If "Customer" and "Account" are two objects in a program then the customer object may send a message to the account object requesting for bank balance without divulging the details of each other's data or code.
- Code in object-oriented programming is organized around objects.

Object- A representation



Object- *Key Goals!!*

Goals of Object definition-

- Define Objects and classes
- Describe objects' methods, attributes and how objects respond to **messages**
- Define Polymorphism, Inheritance, data abstraction, encapsulation, and protocol
- Describe **objects relationships**
- Describe object persistence
- Understand meta-classes.

Object- *An Example*

Example:

Attributes
- I am a Car.
- I know my color,
- manufacturer, cost,
- owner and model.

It does things (methods)
- I know how to
- compute
- my payroll.

Attributes or **properties** describe object's state (data) and **methods** define its **behaviour**.

Object- Attributes and Methods

Object's Attributes
- Attributes represented by data type.
- They describe objects **states**.
- In the Car example the car's attributes are: color, manufacturer, cost, owner, model, etc.

Object's Methods
- Methods define objects behavior and specify the way in which an Object's data are **manipulated**.
- In the Car example the car's methods are: drive it, lock it, carry passenger in it.

Objects- blueprints of classes
- The role of a class is to define the **state** and **behavior** of its instances.
- The class car, for example, defines the property color.
- Each individual car will have property such as "maroon," "yellow"

Classes – *The Blueprint !!*

- *A **class** is a blueprint of an object.*
- A class is a **group of objects** that share **common properties** & **behavior/ relationships**.
- In fact, **objects** are the variables of the **type class**.
- Classes are **user defined data types** and behaves like the built-in types of a programming language.
- **Class** are a **concept**, and the **object** is the embodiment of that **concept**.
- Each class should be designed and programmed to accomplish one, and only one, thing, in accordance to **single responsibility principle** of SOLID design principles.
- In the OOPs concept the variables declared inside a class are known as "**Data Members**" and the functions are known as "**Member Functions**"

Class Members

- A class has different members, and developers in Microsoft suggest to program them in the following order:
- **Namespace**: The namespace is a keyword that defines a **distinctive name** or last name for the class. A namespace categorizes and organizes the library (assembly) where the class belongs and avoids **collisions** with classes that share the same name.
- **Class declaration**: Line of code where the class name and type are **defined**.
- **Fields**: Set of **variables** declared in a class block.
- **Constants**: Set of constants declared in a **class block**.
- **Constructors**: A method or group of methods that contains code to **initialize** the class.

Class Members

- **Properties**: The set of **descriptive data** of an object.
- **Events**: Program **responses** that get fired after a user or application action.
- **Methods**: Set of **functions** of the class.
- **Destructor**: A method that is called when the class is **destroyed**. In managed code, the Garbage Collector is in charge of destroying objects; however, in some cases developers need to take extra actions when objects are being released, such as freeing handles or deallocating unmanaged objects.

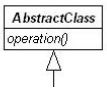
Classes – A Classification

A **Class** "is a set of objects that share a common s
a common behavior." [Booch 1994].

Abstract Classes cannot be instantiated directly.

- The main purpose of an abstract class is to define a comm
  for its subclasses.

Concrete Classes are not abstract and can have ir

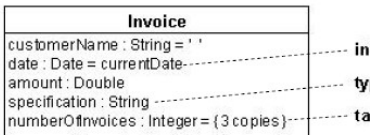


Superclass

Attributes

An **Attribute** is a named data element within
describes the values that instances of the class

- Attributes show the states of an objects with attribu
- Example: an Invoice class



in
ty
ta

Data Abstraction

- **General**: **Focus on the meaning**
  Suppress irrelevant "implementation" details
- It refers to the act of **representing essential features** without including the **background details** or **explanations**.
- Through the process of abstraction, a programmer hides all but the **relevant data** about an object in order to reduce complexity and increase efficiency.
- Abstraction tries to **minimize details** so that the programmer can focus on a **few concepts** at a time. This programming technique separates the **interface** and **implementation**.
- Once you have **modelled** your **object** using Abstraction , the same set of data could be used in **different applications**.

Data Abstraction- *The Motivation*

- **Client/user perspective (Representation Independence)**
  - Interested in **what** a program does, not **how**.
  - Minimize irrelevant details for clarity.
- **Server/implementer perspective (Information Hiding)**
  - Restrict users from making **unwarranted assumptions** about the implementation.
  - Reserve right to **change representation** to improve performance, … (maintaining behavior).

Data Abstraction- *Advantageous!!*

Advantages Of Abstraction

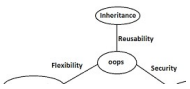
- The programmer does not have to write the **low-level code**.
- The programmer does not have to specify all the **register/binary-level steps** or care about the hardware or **instruction set details**.
- **Code duplication** is avoided and thus programmer does not have to repeat fairly common tasks every time a similar operation is to be performed.
- It allows **internal implementation details** to be changed without affecting the **users** of the abstraction.

Data Encapsulation

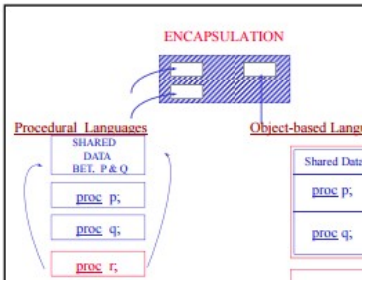
- The **wrapping up** of **data** & **functions** (that operate on the data) into a **single unit** (called class) is known as **ENCAPSULATION**.
- Encapsulation is the **mechanism** that **binds together code** and the **data** it manipulates and keeps both safe from outside interference and misuse.
- Enables **enforcing data abstraction**

Conventions are no substitute for enforced constraints.

- Enables **mechanical detection** of **typos** that manifest as "illegal" accesses. (Cf. problem with global variables).



Data Encapsulation- *An Insight!*



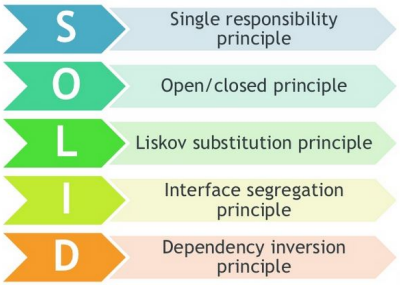
Inheritance

- Inheritance allows the reusability of an **existing operations** and **extending the basic unit** of a class without creating from the scratch.
- Inheritance is the **capability** of one class of things **to inherent properties** from other class.
- Supports the concept of **Hierarchical classification.**
- Ensures the **closeness** with **real world models**.
- Provides **Multiple Access Specifiers** across the **modules** (Public, Private & Protected)
- Supports **Reusability** that allows the addition of **extra features** to an existing class **without modifying it**.

Inheritance : *Sub-classing*

- **Code reuse**
  derive Colored-Window from Window (also adds fields/methods)
- **Specialization: Customization**
  derive bounded-stack from stack (by overriding/redefining push)
- **Generalization: Factoring**
  Commonality – code sharing to minimize duplication – update consistency
- Using two concepts of inheritance, **subclassing** (making a new class based on a previous one) and **overriding** (changing how a previous class works), you can organize your objects into a **hierarchy**.

SOLID Design Principles



SOLID Design Principles

S- A class should have one and only one reason to change, meaning that a class should have only one job.

O- Objects or entities should be open for extension, but closed for modification.

L- All this is stating is that every subclass/derived class should be substitutable for their base/parent class.

I- A client should never be forced to implement an interface that it doesn't use or clients shouldn't be forced to depend on methods they do not use.

D- Entities must depend on abstractions not on concretions. The high level module must not depend on the low level module, but they should depend on abstractions.

Open-Closed Principle

- **Open-closed principle**
  - A class is closed because it can be **compiled**, stored in a **library**, and made available for use by its clients.
- **Stability**
  - A class is open because it can be **extended** by adding **new features (operations/fields)**, or by **redefining inherited features**.

- Inheritance allows the developers for reusing the available code
- A **subclass** can be treated as if it is a **super** class
- Objects of both super class and subclass can be created in the applications
- A class an be extended in which the **additional** and exclusive functionality can be placed without altering the super class
- Relationships among objects can easily be established

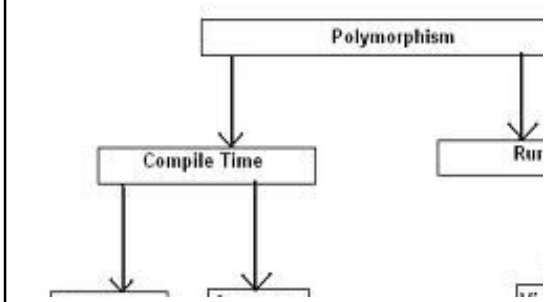
Polymorphism

- In object oriented programming, *polymorphism* refers to a programming language's **ability** to **process objects differently** depending on their data types or class.
- Polymorphism is the **quality** that allows one name to be used for two or more related but **technically different purposes**. In the following, each **graphical object** has the same services, although they are implemented differently.
- If you think about the Greek roots of the term, Polymorphism is the **ability** (in programming) to present the same interface for differing underlying forms (data types).
- For example, **integers** and **floats** are implicitly polymorphic since you can add, subtract, multiply and so on, irrespective of the fact that the types are different. They're rarely considered as objects in the usual term.

Polymorphism

- In object oriented programming, *polymorphism* refers to a programming language's **ability** to **process objects differently** depending on their data types or class.
- Polymorphism is the **quality** that allows one name to be used for two or more related but **technically different purposes**. In the following, each **graphical object** has the same services, although they are implemented differently.
- If you think about the Greek roots of the term, Polymorphism is the **ability** (in programming) to present the same interface for differing underlying forms (data types).
- For example, **integers** and **floats** are implicitly polymorphic since you can add, subtract, multiply and so on, irrespective of the fact that the types are different. They're rarely considered as objects in the usual term.

Polymorphism- *It's Variants!!*



Unified Object Modelling

- The UML effort started officially in October 1994, when Rumbaugh joined **Booch** at **Rational**.
- The Unified Modeling Language (UML) is a **standard language** for writing software blueprints. The UML may be used to **visualize**, **specify**, **construct**, and **document** the artifacts of a software intensive system.
- The UML is appropriate for **modelling systems** ranging from enterprise information systems to distributed Web-based applications and even to hard real time embedded systems.
- The UML is process **independent**, although optimally it should be used in a process that is use case driven, architecture-centric, iterative, and incremental.

Paradigm Shift

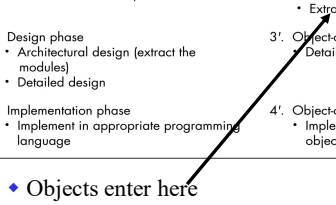
Structured Paradigm	Object-OrientedParad
1. Requirements phase	1. Requirements phase
2. Specification (analysis) phase	2′. Object-oriented analys
3. Design phase	3′. Object-oriented design
4. Implementation phase	4′. Object-oriented progra
5. Integration phase	5. Integration phase
6. Maintenance phase	6. Maintenance phase

Traditional paradigm:
- Jolt between **analysis** (what) and **design** (how)

Object-oriented paradigm:
- Objects enter from **very beginning**

Paradigm Shift

Structured Paradigm	Object-Oriented Paradigm
2. Specification (analysis) phase • Determine what the product is to do	2′. Object-oriented analysis phase • Determine what the product is to do • Extract the objects
3. Design phase • Architectural design (extract the modules) • Detailed design	3′. Object-oriented design phase • Detailed design
4. Implementation phase • Implement in appropriate programming language	4′. Object-oriented programming phase • Implement in appropriate object-oriented programminglanguage



Analysis/Design Analogue

System analysis

- Determine what has to be done
- Determine the **objects**

Design

- Determine how to do it
- Design the **objects**
- **Detailed design**—design each module

Object Oriented Thinking

- Identify all objects in this classroom and articulate their object diagrams. Specify each objects attributes and behaviors through this diagram. Correspondingly identify relationships between the objects.

Benefits of OO Thinking

- Ease to develop complex systems
- Systems are prone to change
- Systems with user interfaces
- Systems that are based on client/servermodel
- To build e-commerce/web based applications
- For enterprise application integration
- Improved quality, reusability, extensibility
- Reduce maintenance burden
- Financial benefits

Challenges in OO Thinking

- Mind-set transition
- Investment in training and tools
- Insist on testing
- More time and cost to analysis and design
- User involvement
- Provides only long term benefits
- Still the success is greatly depends on people involved

Links and Associations

- **Links** and **association** are the means for building the **relationship** among the **objects** and **classes**.
- Links and association , both are quite same feature but **links** establishing among the **objects** (instance) and **association** establishing among the **class**.

  "Link is related to objects whereas association is related to classes"
- **Class diagrams** contain **associations**, and **object diagrams** contain **links**.
- Both associations and links represent **relationships**.
- Links as well as associations appear as **verbs** in a problem statement.

Links and Associations

•Can link and Association applied interchangeably?

- No, You cannot apply the link and Association interchangeably.
- Since **link** is used represent the **relationship** between the **two objects.**
- But **Association** is used represent the **relationship** between the **two classes.**

Link :: student:Abhilash course:MCA
Association:: student course

Links

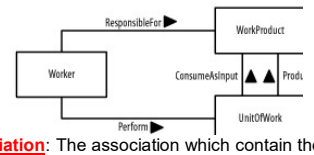
- In **object modelling** links provides a **relationship** between the **objects**.
- These objects or instance may be same or different in **data structure** and **behaviour**.
- Therefore a link is a **physical or conceptual connection** between instance (or **objects**).
- For **example**: Ram works for HCL company. In this example "**works for**" is the link between "Ram" and "HCL company". Links are relationship among the objects(instance).
- **Types of links**:
  1.One to one links
  2.one to many and many to one links
  3. many to many

Associations

- The object modelling describes as a **group of links** with **common structure** and **common semantics**.
- "***Association*** *is a* ***relationship*** *between* ***classifiers*** *which is used to show that instances of classifiers could be either* ***linked*** *to each other or combined* ***logically*** *or* ***physically*** *into some aggregation.*"
- All the links among the object are the forms of **association** among the **same classes**.
- The association is the **relationship** among **classes**.
- UML specification categorizes association as **semantic relationship**. Some other UML sources also categorize association as a **structural relationship**. Wikipedia states that association is **instance level** relationship and that associations can only be shown on **class diagrams**.

Degree of Association

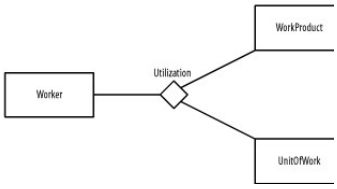
- **Unary Association**: the association can be defined on a **single class**. This type of association called unary (or **singular**) **association**.
- **Binary Association**: The binary association contain the **degree of two classes**. The association uses two class.



- **Ternary Association**: The association which contain the **degree of three classes** is called **ternary association**. The ternary Association is an atomic unit and cannot be subdivided into binary association without losing information.

Degree of Association

- **Quaternary Association**: The Quaternary Association exists when there are **four classes associated**.
- **Higher degree Association**: The higher order association are more **complicated** to draw , implement because when **more than four class** need to be associated then it seems a hard task.



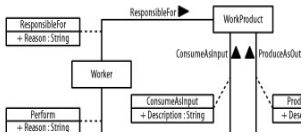
Association Classes

- 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 **behavioural features**, an *association class* may be used to define the **characteristics** of its **links**, including their **structural features** and behavioural 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**.

Binary Association Classes

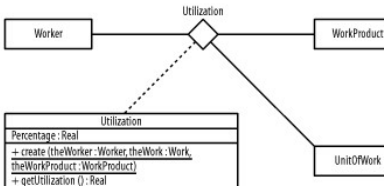
The following example shows **association classes** for the binary associations in the most basic notation for binary association classes.

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.



n-ary Association Classes

- The following example shows an association class for the n-ary association in 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.



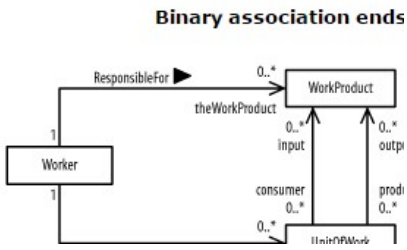
Association Ends

- 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:
  - ✓Role name
  - ✓Navigation arrow
  - ✓Multiplicity specification
  - ✓Aggregation or composition symbol
  - ✓Qualifier

I. 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 the figure

I. Rolenames

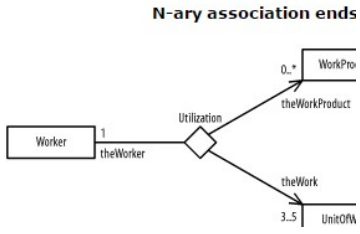


II. Navigation

- *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, arrows pointing towards work product and units of work. Given a unit of work, you can determine its input and output work products

II. Navigation

- 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.



III. 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.

III. Multiplicity Indicators

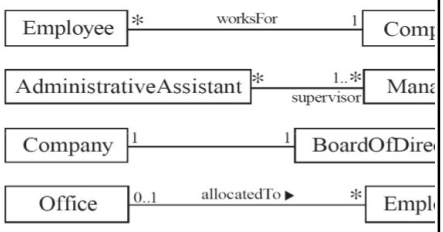
- Unspecified
- Exactly one — 1
- Zero or more (many, unlimited) — 0..*
- One or more — 1..*
- Zero or one (optional scalar role) — 0..1
- Specified range — 2..4
- Multiple, disjoint ranges — 2, 4..6

III. Multiplicity

- Multiplicity is the **number of instances** of one class **relates to instance of another class.**
- For the following association, there are two multiplicity decisions to make, one for each end of the association.
  - For each instance of Professor, many Course Offerings may be taught.
  - For each instance of Course Offering, there may be either one or zero Professor as the instructor.

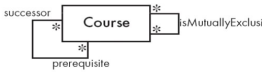
Labelling Associations

- Each association can be labelled, to make nature of the association

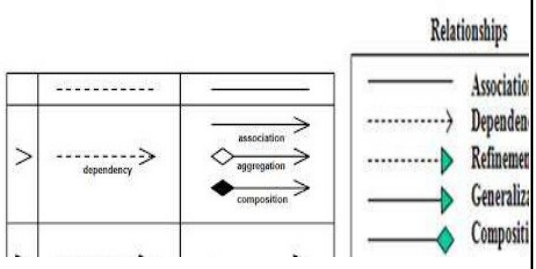


Reflexive Associations

- It is possible for an **association** to **connect a class to itself.**
- There are two main types:
  - **Symmetric** and **Asymmetric**.
- **Asymmetric Reflexive Associations:** The ends of the association are semantically different from each other, even though the associated class is the same. Examples include parent-child, supervisor-subordinate and predecessor-successor.
- **Symmetric Reflexive Associations:** There is **no logical difference** in the semantics of each association end. In other words, students who have taken one course cannot take another in the set. Umple uses the keyword **'self'** to identify this case.

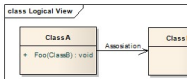


Association Nomenclature

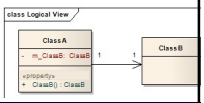


Association- *Further more!*

- The most abstract way to describe static relationship between classes is using the **Association** link, which simply states that there is ***some kind of a link or a dependency*** between two classes or more.
- **Weak Association** - ClassA may be linked to ClassB in order to show that one of its methods includes **parameter** of ClassB instance, or returns instance of ClassB.



- **Strong Association** - ClassA may also be linked to ClassB in order to show that it holds a **reference** to ClassB instance.

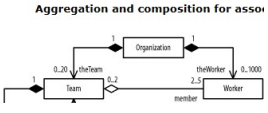


Association- *Further more!*

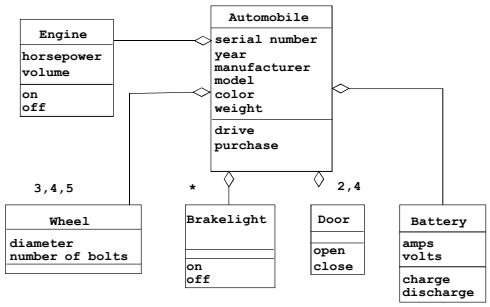
- In **Object-oriented programming**, one object is related to other to use functionality and service provided by that object.
- This relationship between two objects is known as the *association* in object oriented general software design and depicted by an arrow in Unified Modelling language or UML.
- Both **Composition** and **Aggregation** are the form of **association** between two objects, but there is a **subtle difference between composition and aggregation**, which is also reflected by their UML notation.
- *The composition is stronger than Aggregation*.
- In Short, a **relationship** between two objects is referred as an **association**, and an **association** is known as **composition** when ***one object owns other*** while an **association** is known as **aggregation** when *one object uses another object*.

Aggregation *(Shared Association)*

- Aggregation is **whole-part relationship** between an **aggregate**, the whole, and its **parts** where the part can exist **independently** from the aggregate.
- This relationship is often known as a *has-a relationship*, because the **whole has its parts**.
- Aggregation is shown using a **hollow diamond** attached to the class that represents the **whole**.
- Creating a **circular relationship** to allow for sub-teams is known as a reflexive relationship, because it relates two objects of the same class.



Aggregation Scenario



Aggregation- *When to use!*

- As a general rule, you can mark an association as an aggregation if the following are true:

 You can state that

  The parts 'are part of' the aggregate
  The aggregate 'is composed of' the parts

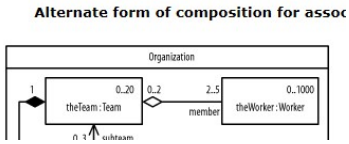
 When something owns or controls the aggregate, then they also own or control the **parts**.

Composition *(Non-shared)*

- 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**.
- Composition is shown using a **filled diamond** attached to the class that represents the whole.
- For example, an organization contains teams and workers, and if the organization ceases to exist, its teams and workers also cease to exist.
- 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.

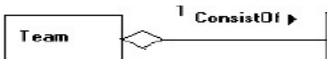
Composition

- Separate the **rolename** from the **class** name using a **colon**.
- A composition indicates a **strong ownership** and **coincident lifetime** of parts by the whole (i.e., *they live and die as a whole*).



Aggregation vs. Composition

Example: Aggregation



Aggregation vs. Containment

- **Aggregation** is the relationship between the whole and a part. We can add/subtract some properties in the part (slave) side. It won't affect the whole part.
- Best **example** is Car, which contains the wheels and some extra parts. Even though the parts are not there we can call it as car.
- But, in the case of **containment** the whole part is **affected** when the part within that got affected.
- The **human body** is an apt example for this relationship. When the whole body dies the parts (heart etc.) are dead.

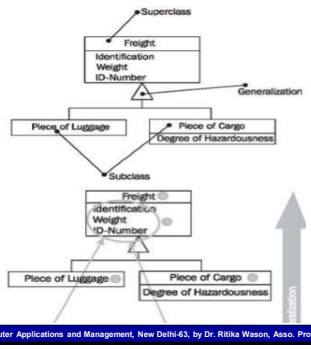
System Complexity Measure

- **System complexity** can be measured simply by looking at a UML class diagram and **evaluating the association**, **aggregation**, and **composition relationship lines**.
- The way to measure complexity is to determine *how many classes can be affected by changing a particular class*.
- If class *A* exposes class *B*, then any given class that uses class *A* can theoretically be affected by changes to class *B*.
- The **sum** of the number of **potentially affected classes** for every class in the system is the total system complexity.

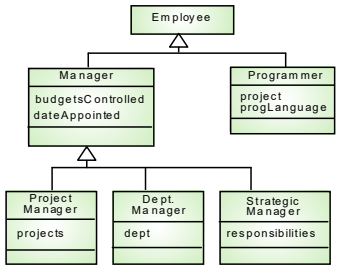
Generalization

- *Generalization is the process of extracting shared characteristics from two or more classes*, and **combining** them into a **generalized superclass**.
- Shared characteristics can be **attributes**, **associations**, or **methods**.
- Generalization is a process of **defining** a super class from a given *set of semantically related entity set*.
- Generalization uses a ***"is-a" relationship*** from a **specialization** to the **generalization class**.
- Common **structure** and **behaviour** are used from the specialization to the generalized class.

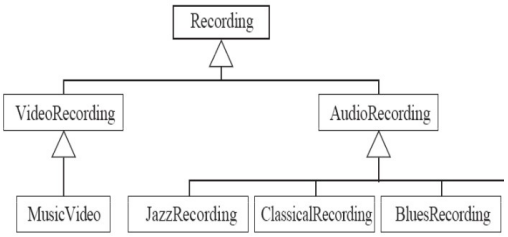
Generalization Example



Generalisation Hierarchy

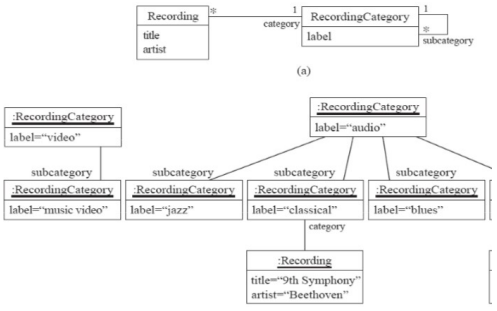


Avoid unnecessary Generalization



Inappropriate hierar

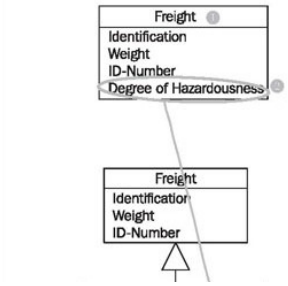
Improvised Hierarchy



Specialization

- *Specialization* means creating **new subclasses** from an **existing class**.
- If it turns out that certain **attributes**, **associations**, or **methods** only apply to some of the objects of the class, a *subclass* can be created.
- The most **inclusive class** in a generalization/specialization is called the **superclass** and is generally located at the **top** of the diagram.
- The more **specific classes** are called **subclasses** and are generally placed **below** the superclass.

Specialization Example



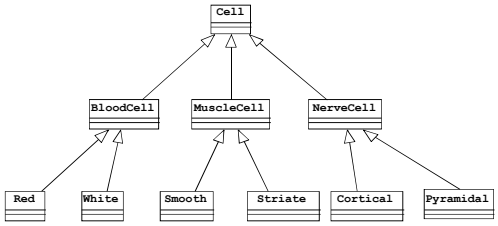
Inheritance

• The **generalization/specialization** relationship is implemented in object oriented programming languages through **inheritance**.

- Object-oriented programming allows classes to ***inherit*** commonly used **state** and **behaviour** from other classes.
- A class that is **derived** from another class is called a ***subclass*** (also a *derived class*, *extended class*, or *child class*). The class from which the subclass is derived is called a ***superclass*** (also a *base class* or a *parent class*).
- When you want to create a new class and there is already a class that includes some of the code that you want, you derive it.
- A subclass inherits all the ***members*** (fields, methods, and nested classes) from its superclass.
- Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass.

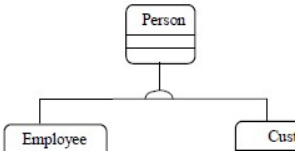
Inheritance

- Models "kind of" hierarchy
- Powerful notation for sharing similarities among classes while preserving their differences
- UML Notation: An **arrow with a triangle**



Inheritance Hierarchy

- The **generalized** class at the top and the **specialized** classes below.
- The specialized class names should reflect the class they were **specialized from**.
- For example, employee was specialized from Person.



Aggregation vs Inheritance

- Both associations describe trees (hierarchies)
  - **Aggregation** tree describes a-part-of relationships (also called **and-relationship, Has –a Relationship, containership**)
  - **Inheritance** tree describes "**kind-of**" relationships (also called **or-relationship, is-a relationship**)
- Aggregation **relates instances** (involves two or more *different* **objects**)
- **Inheritance relates classes** (a way to structure the description of a *single* object)

Realization

- Realization is a **relationship** between the **blueprint class** and the **object** containing its respective ***implementation level details***.
- This object is said to *realize the blueprint class*.
- In other words, you can understand this as the relationship between the **interface** and the **implementing class.**

- **Example**: A particular model of a car 'GTB Fiorano' that implements the blueprint of a car realizes the abstraction.

Dependency

- Change in **structure** or **behaviour** of a class affects the other **related class**, then there is a *dependency* between those two classes. It need not be the same vice-versa.
- When one class **contains** the other class it this happens.

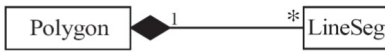
- ***Example:*** Relationship between shape and circle is dependency

Dependency

- It is the relationship between **dependent** and **independent classes**.
- Any change in the independent class will *affect the states* of the dependent class.
- A dependency is a relation between two classes in which a change in ***one may force changes*** in the other although there is no explicit association between them.
- A stereotype may be used to denote the type of the dependency.
  - Indicates a semantic relationship between **two (or more) classes**
  - It indicates a **situation** in which a change to the target element may require a change to the source element in the dependency
  - A dependency is shown as a **dashed arrow** between two model elements

Propagation

- A mechanism where an **operation** in an **aggregate** is implemented by having the aggregate perform the operation on its **parts**.
- At the same time, **properties** of the **parts** are often **propagated** back to the **aggregate**.
- *Propagation is to aggregation as inheritance is to generalization.*
- The major difference is-
  - Inheritance is an implicit mechanism
  - Propagation has to be programmed when required.



Constraints

- To **restrict** ways in which a class can **operate** we add constraints.
- **OCL** is a **specification language** designed to formally specify constraints in software modules.
- Types of constraints
  - Invariant
    - ✓ Must be always true
    - ✓ Defined on class attributes
  - Pre-condition
    - ✓ Defined on a method
    - ✓ Checked before execution
    - ✓ Frequently used to validate input parameter
  - Post-condition
    - ✓ Defined on a method
    - ✓ Checked after method execution
    - ✓ Frequently used to describe how values were changed by method

Constraints

- Constraint defines some **functional relationship** between **entities** of an **object**.
- The term **entity** includes objects , classes , attributes , links and association.
- It mean constraints can be implemented on the **objects**, **classes** , **attributes** , **links** as well as on **association** too.

Think about it!

• For the following cases, indicate whether the relationship should be an ordinary association, a standard aggregation, a composition, a dependency, a Realization. Justify your answer.

a. Student taught by teacher
b. Department has Teachers
c. House and Rooms
d. Person and electric switch (to start a fan).
e. Code snippet

```
import B;
public class A { public void method1(B b) { // . . . }..}
```

f. Code snippet

```
import B3;
public class A3 implements B3 {   // . . .}
```


Suggested Sequence

- Identify a first set of **candidate classes**
- Add **associations** and **attributes**
- Find **generalizations**
- Find **specializations**
- List the **main responsibilities** of each class
- Decide on specific **operations**
- **Iterate** over the **entire process** until the model is satisfactory
  - Add or delete classes, associations, attributes, generalizations, responsibilities or operations
  - Identify interfaces
- Don't be too **disorganized**.
- Don't be too **rigid** either.

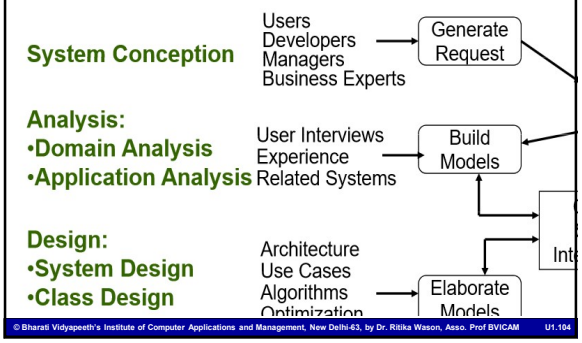
Object Oriented Lifecycle Model

- Object Oriented Methodology (OOM) is a **system development approach** encouraging and facilitating **re-use** of **software components**.
- The object-oriented systems analysis and design methodology classification emerged in the **mid- to late 1980s** as businesses began to seriously consider object-oriented-programming languages for developing and implementing systems.
- 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, and even an account is an object.

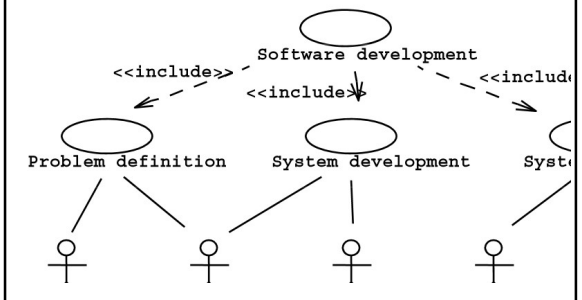
OOLCM - *Representation*

	TECHNIQUES & TOOLS REPRESEN		
	System Flows	Data	Process Logic
TRADITIONAL	System Flowchart	Forms, Layouts, Grid Charts	English Playscrip Program HIPO
STRUCTURED	Data Flow Diagram	Data Dictionary, Data Structure Diagrams, E-R Diagrams	Decision Structure Structure Warnier/
DATA MODELING (INFO RMATION ENGINEERING)	Business Area Analysis, Process Model	Business Area Analysis, E-R Diagrams	Business Design
OBJECT-	Object Model	Object Model	Static &

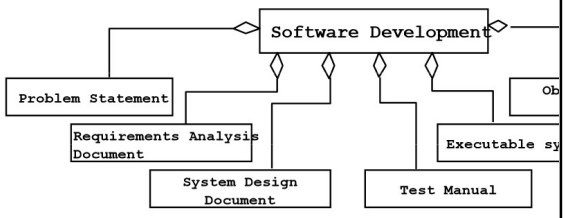
Object Oriented Lifecycle Model



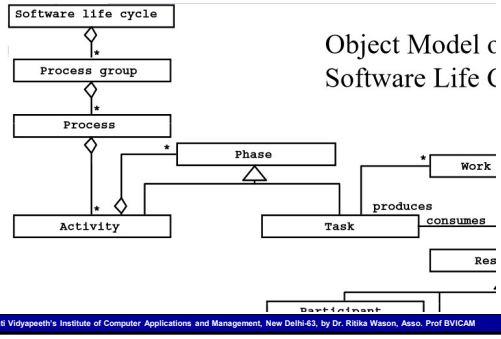
Software Development- Usecase Model



Software Development- Simple Object Model



Software Development- Object Model



OOLCM- Analysis

- Object Modelling is based on **identifying** the **objects** in a system and their **interrelationships**.
- As in any other system development model, system analysis is the **first phase** of development in case of Object Modelling 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**.

OOLCM- 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**.

OOLCM- Design

- 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 Orientation in 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.
- **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*.

Object Orientation in Design

- 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**.

Object Orientation in Design

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.

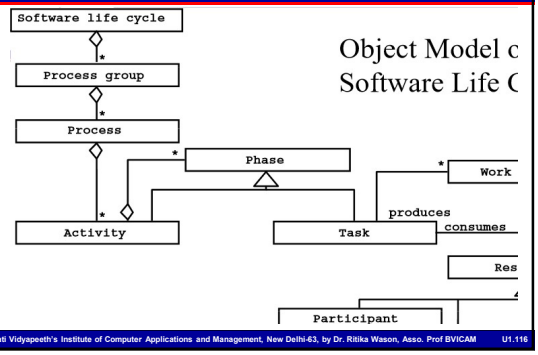
OOLCM- Implementation

- 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 **behaviour**.
- 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.

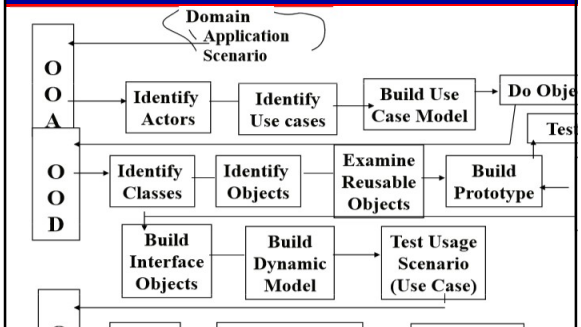
OOLCM- Implementation

- 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 analyse 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.

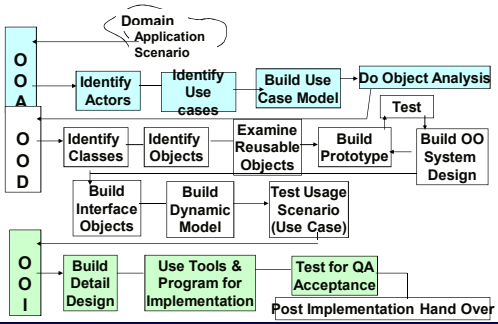
OOLCM- *Detailing!*



OOLCM – *Step-wise flow*



Life Cycle Model OOSAD



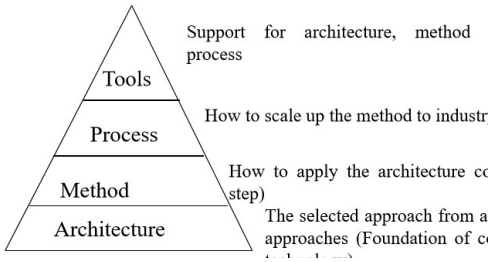
Object Orientation- *Advantages*

- 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**.

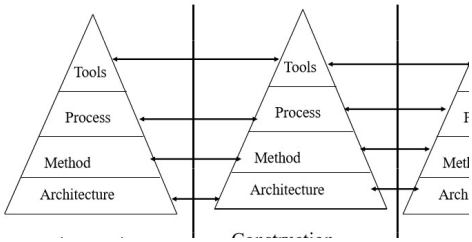
Software Development- Industrial Process

- Process must yield a **foreseeable result**, **irrespective** of which **individual** performed the job.
- **Volume of output** doesn't effect the **process**.
- Possibility to **allocate parts** of the **process** to several **manufacturers** and **subcontractors**.
- Possible to make use of **pre-defined building blocks** and **components**.
- Possible to **plan** and **calculate** the process with **great precision**.
- Each **person** trained for an **operation** must perform it in a **similar manner.**

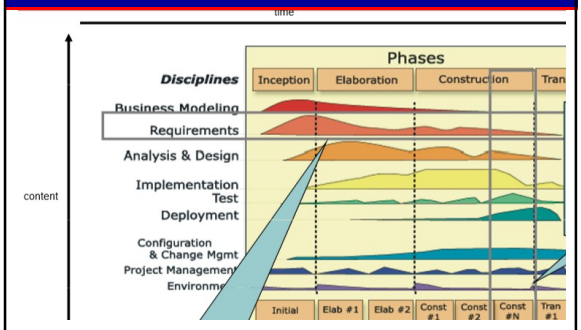
Rational Enterprise Philosophy



Multiple Activities- R.E.



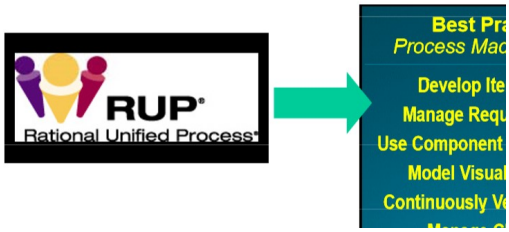
Unified Approach



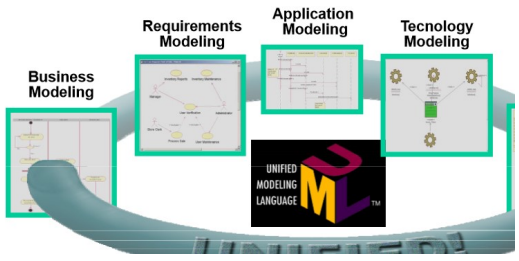
Unified Approach

- Based on the best practices
- Unify the modeling efforts of Booch, Rumbaugh and Jacobson
- Revolves around the processes and concepts
  - Use-case driven development
  - Object Oriented Analysis
  - Object oriented Design
  - Incremental development and prototyping

RUP implements best practices



One language for all practitioners



Unified Approach Phases

Has four phases

- Inception
  - ✓Understand problem
- Elaboration
  - ✓Understand Solution
- Construction
  - ✓Have a Solution
- Transition

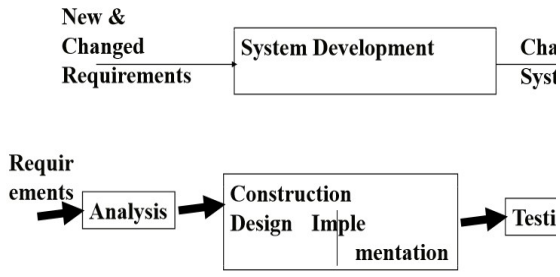
Disciplines of RUP

1. Business Modelling
2. Requirements
3. Analysis and Design
4. Implementation
5. Test
6. Deployment

System Development Characteristics

- ❑ Part of a larger activity
- ❑ System development
- ❑ Transition from analysis to construction
- ❑ Requirements are inputs to system development
- ❑ A system is output system development
- ❑ Parties interested in system development are customer, direct and indirect users, etc.

System Development



Responsibility-Driven Design

- In Responsibility-Driven Design, a model is developed from the **requirements specification** by the **extraction** of **nouns** and **verbs** from the specification.
- This provides a basis for the **actual implementation.**
- In RDD, for each class, **different responsibilities** are defined which specify the **roles of the objects,** and their **actions.**
- In order to fulfill these responsibilities, classes need to **collaborate** with each other. Collaborations are defined to show *how the objects will interact.*
- The responsibilities are further grouped into **contracts** which define a **set of requests** that objects of the class can support.
- These contracts are further refined into **protocols**, which show the **specific signature** of each operation.

Responsibility-Driven Design

- The RDD modelling process includes two **phases**:
- **Exploratory**: The exploratory phase has three goals-- finding the **classes**, determining **responsibilities**, and identifying **collaborations**. This is commonly done with the CRC Cards.
- **Analysis**: The analysis phase involves refining the **object's behaviour** and the **service definitions** specified in the exploratory phase. These activities include defining **interfaces** (protocols) and constructing **implementation specifications** for each class.

Class Responsibility Collaboration

Classes
- Extract **noun phrases** from the **specification** and build a list
- Identify candidates for abstract **super classes**
- Use categories to look for missing classes
- Write a short statement for the **purpose** of each **class**

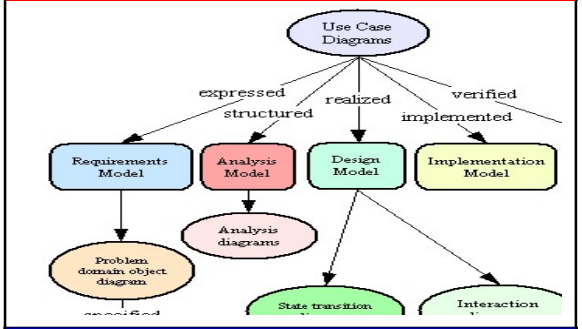
Responsibilities
- Find **responsibilities**
- Assign responsibilities to **classes**
- Find additional responsibilities by looking at the **relationships** between classes

Collaborations
- Find and list **collaborations** by examining responsibilities associated with classes
- Identify additional collaborations by looking at **relationships between classes**
- **Discard** and classes that take part in no collaboration (as client or server)

Object-Oriented Software Engineering

• Object-oriented software engineering (OOSE) is an **object modeling language** and **methodology**.
• 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. OOSE is one of the precursors of the Unified Modeling Language (UML), such as Booch and OMT.
• It includes a **requirements**, an **analysis**, a **design**, an **implementation** and a **testing** model.
• **Interaction diagrams** are similar to UML's sequence diagrams. State transition diagrams are like UML **statechart diagrams.**

Object-Oriented Software Engineering



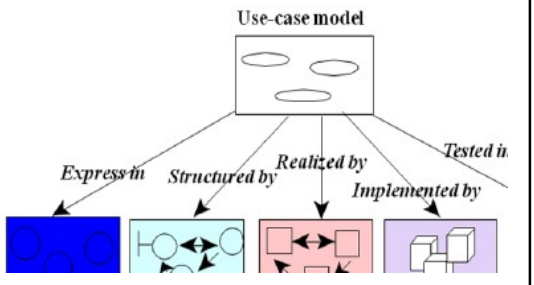
Jacobson OOSE

- Object-Oriented Software Engineering (OOSE) is a **software design technique** that is used in **software design** in object-oriented programming.Originated from **Objectory** (Object Factory for software development)
- OOSE is developed by **Ivar Jacobson** in 1992. OOSE is the first object-oriented design methodology that employs use cases in software design. OOSE is one of the **precursors** of the Unified Modeling Language (**UML**), such as Booch and OMT.
- It includes a **requirements**, an **analysis**, a **design**, an **implementation** and a **testing** model.
- *Interaction diagrams* are similar to UML's *sequence diagrams*. State transition diagrams are like UML *statechart diagrams*.

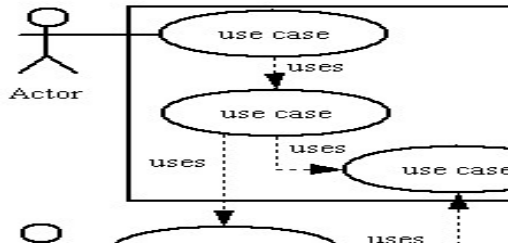
Jacobson OOSE

- Aim to fit the development of **large real-time system**
- **Stress traceability** among the different phases (Backward & forward)
- Supports OO concepts of **classification**, **encapsulation** and **inheritance**.
- Abstraction is promoted by levels.
- Adds "*use cases*" to the OO approach.
- **Composite data** and **activity definition** is not strongly enforced and services are also regarded as **objects**.
- **Reuse** is supported by **component libraries**.
- Guidance for analysis is less comprehensive.
- Target applications: like **HOOD real-time systems** and **engineering systems**.

Jacobson's Use Case Model



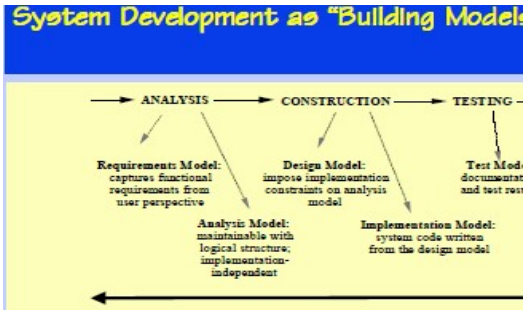
Jacobson's Use Case Diagram



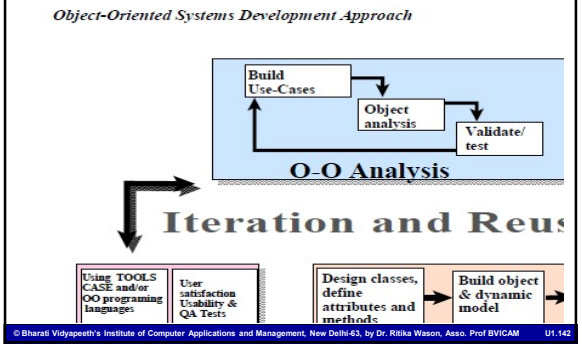
Objectory

- **Discipline process** for the industrialized development of software, based on a *use case driven design*
- Built around several different models
  - Requirement Model
  - Domain object model
  - Analysis Model
  - Design Model
  - Implementation model
  - Test model

OOSE Models



Object Oriented System Dev



OOSE- Requirement Model

- Two different models are developed in **OOSE;** the **Requirements Model** and the **Analysis Model.**
- These are based on **requirement specifications** and discussions with the **prospective users**.
- The first model, the **Requirements Model**, should make it possible to ***define the system*** and to define what functionality should take place within it.
- For this purpose we develop a **conceptual picture** of the system using problem **domain objects** and also **specific interface descriptions** of the system if it is meaningful for this system.
- We also describe the system as a **number of use cases** that are performed by a number of actors.

OOSE- Analysis Model

- The Analysis Model consisting of various **object classes**: ***control object, entity objects,*** and ***interface objects***.
- The purpose of this model is to find a **robust and extensible structure** for the **system** as a **base for construction**.
- Each of the **object types** has its own **special purpose** for this robustness, and together they will offer the total functionality that was specified in the Requirements Model.

OOSE- Construction

- We build our system through **construction** based on the Analysis Model and the Requirements Model created by the analysis process.
- The construction process lasts until the **coding** is completed and the included units have been tested.
- There are three main reasons for a construction process:
  - *1) The Analysis Model is not sufficiently formal.*
  - *2) Adaptation must be made to the actual implementation environment.*
  - *3) We want to do internal validation of the analysis results.*

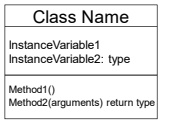
OOSE- Construction

- The construction activity produces two models, the **Design Model** and the **Implementation Model**.
- Construction is thus divided into two phases; **design** and **implementation**, each of which develops a model.
- The **Design Model** is a further **refinement** and **formalization** of the Analysis Model where consequences of the implementation environment have been taken into account.
- The **Implementation model** is the actual **implementation** (code) of the system.

OOSE- Testing

- Testing is an activity to **verify** that a **correct system** is being built.
- Testing is traditionally an **expensive activity**, primarily because many faults are not detected until late in the development.
- To do effective testing we must have as a **goal** that every test should detect a fault

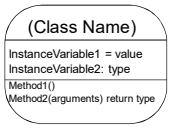
Object Model Notation



Classes are represented as rectangles;

The class name is at the top, followed by attributes (instance variables) and methods (operations)

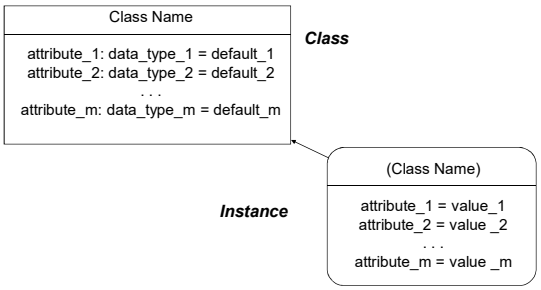
Depending on context some information can be hidden such as types or method arguments



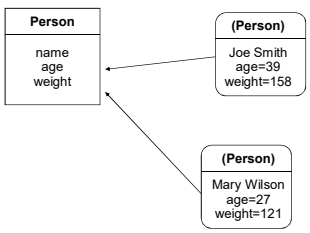
Objects are represented as rounded rectangles;

The object's name is its classname surrounded by parentheses
Instance variables can display the values that they have been assigned; pointer types will often point (not shown) to the object being referenced

OMT Instantiation Notation



Instantiation - Example



Inheritance

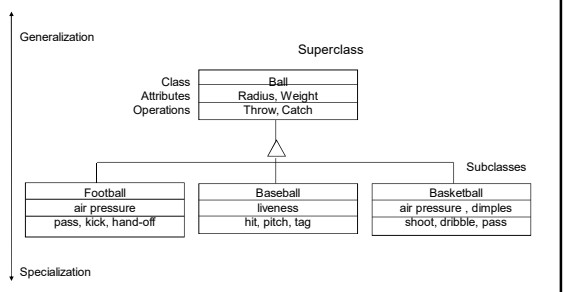
Classes with similar **attributes** and operations may be organized **hierarchically**

Common attributes and operations are factored out and assigned to a broad superclass (*generalization*)

- Generalization is the "**is-a**" relationship
- Super classes are ancestors, subclasses are descendants

Classes iteratively refined into subclasses that *inherit* the attributes and operations of the superclass (*specialization*)

OMT Inheritance Notation



Association and Links

An **association** **is a relation among two or more classes** describing a group of links, with common structure and semantics

A **link** **is a relationship or connection between objects** and is an instance of an association

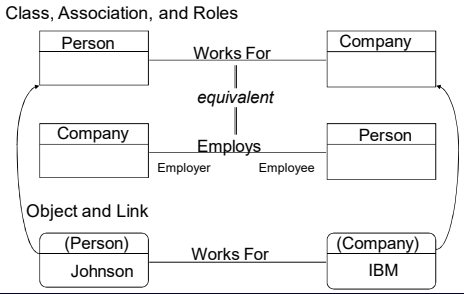
A link or association is inherently *bi-directional*

- the name may imply a direction, but it can usually be inverted
- the diagram is usually drawn to read the link or association from left to right or top to bottom

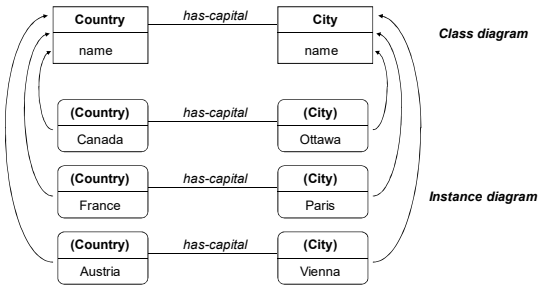
A role is one end of an association

- roles may have names

OMT Association Notation



Association and Links



Aggregation

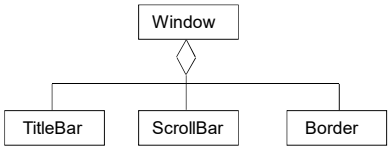
Aggregation is a special form of association that indicates a **"part-of"** relationship between a whole and its parts

Useful when the parts do not have **independent existence**

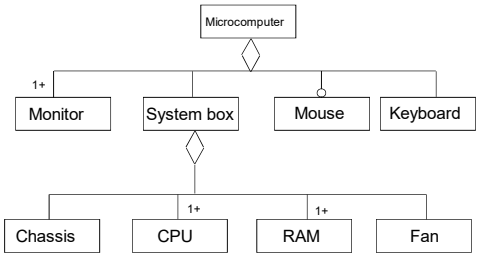
- A part is ***subordinate*** to the whole

In an aggregation, properties and operations may be propagated from the whole to its parts

OMT Aggregation Notation



OMT Multilevel Aggregation



States and Transitions

A *state* is an **interval** between **events** (values of relevant variables to the problem…)

- it may have an activity that can trigger starting, intermediate and ending events
- defined in terms of a subset of object attributes and links

A state *transition* is a change in an object's attributes and links

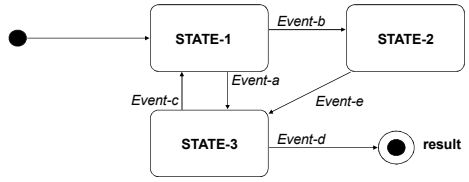
- it is the response of an object to an event
- all transitions leaving a state must correspond to distinct events

OMT State Notation

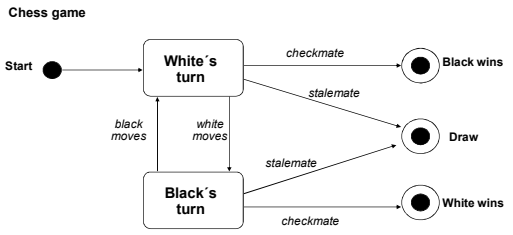
tates represented as nodes: **rounded rectangles with state name**

- initial state represented as solid circle
- final state represented as bull's eye

transitions represented as edges between nodes and labeled with an *event name*



OMT State Diagram - Example



Guards, Activities and Actions

Guards are Boolean conditions on attribute values

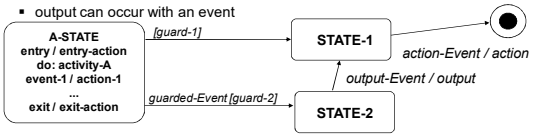
- transition can only happen when guard evaluates to "true"
- automatic transitions occur as soon as an activity is complete (check guard!)

Activities take time to complete

- activities take place within a 'state'

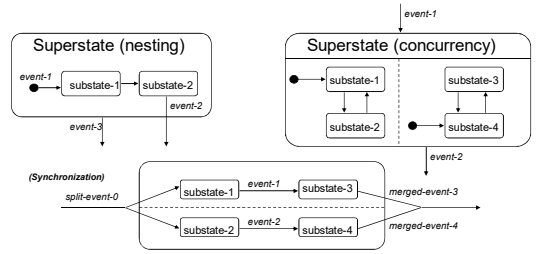
Actions are relatively instantaneous

- actions take place on a transition or within a state (entry, exit, event actions)
- output can occur with an event

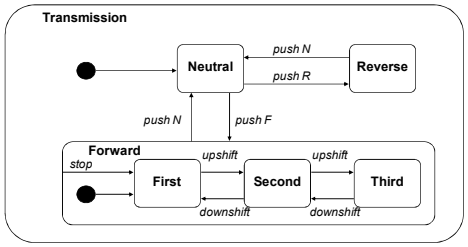


OMT State Relationships

States can be nested or concurrent
Events can be split and merged



State Generalization: example



Structured vs. Unified Process

Criteria	*Structured Methodology*	*Object Oriented (Unified Process)*
Use of development activities (Planning, Analysis..)	Each activity covers a whole phase in SDLC	All activities run in each phase, N-times (iterations)
Names of development phases	Planning, Analysis, Design, Implementation, Installation/Testing	Inception, Elaboration, Construction, Transition
Appropriate to use	When system goals certain, static IT	When system goals less certain, dynamic IT
Modeling technique	Data Flow Diagrams, Entity-Relationship Diagrams	Diagrams defined by *Unified Modeling Language* (Use Cases, Class Diagrams…)
Relation to reality	Predictive	Adaptive

Unified Approach

•UA based on methodologies by Booch, Rumbaugh and Jacobson tries to combine the **best practices**, **processes** and **guidelines** along with the object management groups in unified modelling language.

•UA utilizes the *unified modeling language* (UML) which is a set of notations and conventions used to describe and model an application.

Goals:

- Define **Objects** and **classes**
- Describe objects' **methods**, **attributes** and how objects respond to messages
- Define **Polymorphism**, **Inheritance, data abstraction, encapsulation**, and **protocol**
- Describe **objects relationships**

Object-Oriented Software Dev

Object-Oriented Methodology

- Development approach used to build **complex systems** using the concepts of object, class, polymorphism, and inheritance with a view towards ***reusability***
- Encourages software engineers to think of the problem in terms of the **application domain** early and apply a **consistent approach** throughout the entire life-cycle

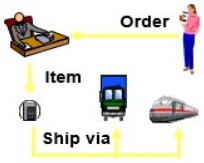
Object-Oriented Analysis and Design

- Analysis models the "*real-world*" requirements, independent of the implementation environment
- Design applies object-oriented concepts to develop and communicate the **architecture** and details of how to meet requirements

Visual Modelling

- **Mapping** **real-world process** of a **computer system** with a ***graphical representation*** is called visual modelling.
- Visual Modeling is a ***way of thinking*** about **problems** by using **graphical models** of **real-world ideas.**

Visual Modelling



"Modeling captures
parts of th
Dr. Jan

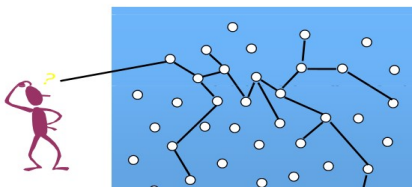
Visual Modeling is
modeling

Benefits of Visual Modelling

- Captures Business Process
- Enhance Communication
- Manage Complexity
- Define Architecture
- Enable Reuse

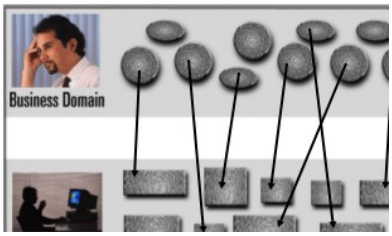
I. Capture Business Processes

- When we create **use cases,** visual modeling allows us to **capture business processes** by defining the ***software system requirements*** from the **user's perspective.**

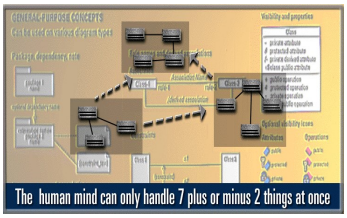


II. A Communication Tool

Use visual modeling to capture business objects



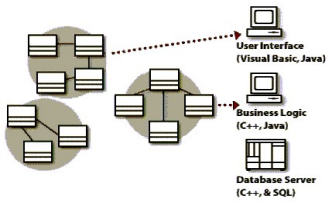
III. Manages Complexity



Systems today typically have hundreds or even thousands of classes. These classes must be organized in such a way as to allow viewing by many different groups of people; often with their own viewing needs.

Visual modeling provides the capability to display modeling elements in many ways, so that they can be viewed at different levels of abstraction.

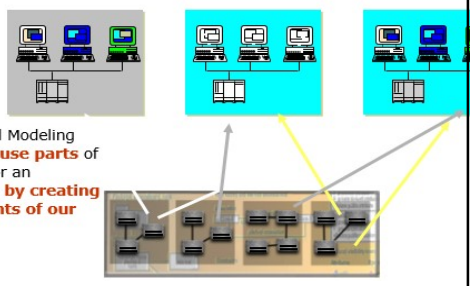
IV. Defines Software Architecture



Visual modeling provides the **capability** to capture the **logical software architecture** **independent** of the **implementation language**.

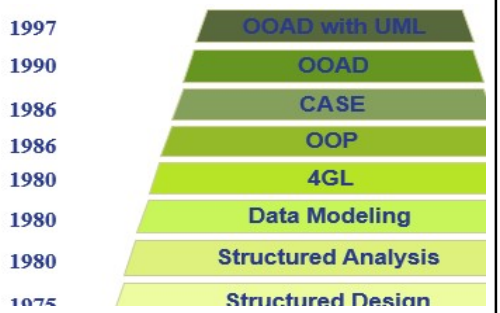
As **system design** progresses, the **implementation language** is determined and the **logical architecture** is **mapped** to the **physical architecture**.

V. Promotes Re-use



With Visual Modeling **we can reuse parts** of a system or an application **by creating components of our design**.

Design Method Evolution



UML Origin

A product of the "**design wars**" of the 1980's

Grady Booch, James Rumbaugh, and others had competing styles.

`94: Rumbaugh leaves GE to join Booch at Rational Software

"Method wars over. We won." Others feared achieving standardization the Microsoft way.

'95: Rational releases **UML 0.8**; Ivars Jacobson (use cases) joins Rational→"The Three Amigos"

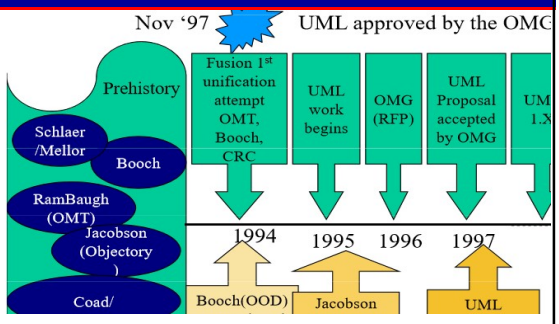
'96: Object Management Group sets up task force on methods

'97: Rational proposed **UML 1.0** to **OMG**. After arm twisting and merging, **UML 1.1** emerges

'99: After several years of revisions and drafts, **UML 1.3** is released

Now **UML 1.5**….

UML Origin



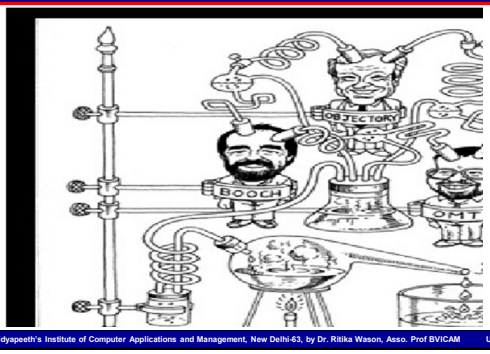
What is UML

- "***The Unified Modeling Language is a family of graphical notations, that help in describing and designing software systems, particularly software systems built using the object-oriented style.***"
- UML first appeared in **1997**
- UML is **standardized**. Its content is controlled by the Object Management Group (OMG), a group of companies.
- UML can be applied to diverse **application domains** (e.g., banking, finance, internet, aerospace, healthcare, etc.) It can be used with all major object and component **software development methods** and for various **implementation platforms** (e.g., J2EE, .NET).

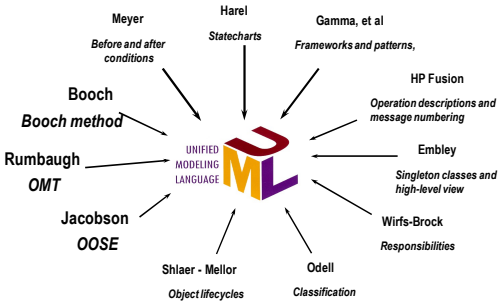
What is UML

- UML stands for **Unified Modelling Language**
- The UML combines the best of the best from
  - Data Modelling concepts (Entity Relationship Diagrams)
  - Business Modelling (work flow)
  - Object Modelling
  - Component Modelling
- The UML is the **standard language** for **visualizing**, **specifying**, **constructing**, and **documenting** the artifacts of a *software-intensive system*
- It can be used with all **processes**, throughout the **development life cycle,** and across different **implementation technologies**

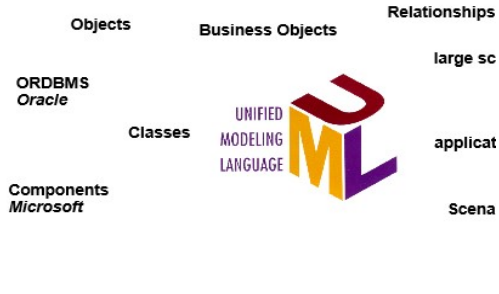
How it all begun…



UML Contributors



UML Supports



Goals of UML

1) Provide users with a **ready-to-use, expressive** visual modeling language

2) Provide **extensibility** and **specialization** mechanisms to **extend** the **core concepts**.

3) Be **independent** of particular **programming languages** and **development processes**.

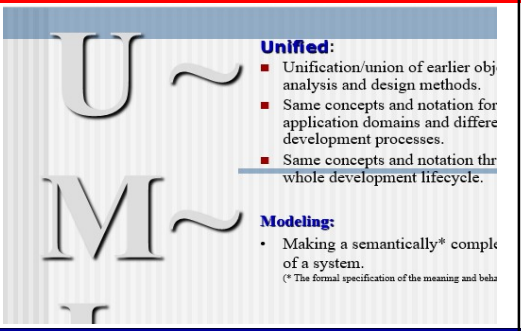
4) Provide a *formal basis for understanding* the modeling language.

5) Encourage the **growth** of the **OO tools market.**

6) Support **higher-level development concepts** such as **collaborations**, **frameworks**, **patterns** and **components**.

7) **Integrate best practices.**

UML



UML- A language for…

- The UML is a language for
  - **visualizing**
  - **specifying**
  - **constructing**
  - **documenting**

  a software-intensive system

- UML can also be applied outside the **domain** of software development.

UML- A language for…

1. Visual Modelling

'A picture is worth a thousand words'

- Use standard **graphical notations**
- **Semi-formal**
- Captures **business processes** from **enterprise information systems** to **distributed web-based applications** and even to hard **real time embedded systems.**

UML- A language for…

2. Specifying

•**Building models** that are
- Precise
- Unambiguous
- Complete

•**Symbols** are based upon
- Well-defined syntax
- semantics

•Addresses the **specification** of all important **analysis**, **design** and **implementation decisions**.

UML- A language for…

3. Constructing

•Models are related to **object oriented programming languages**

•**Round-trip engineering** requires tools and human invention to information loss.

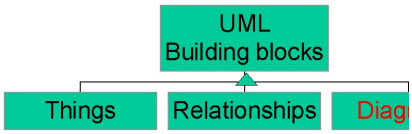
- **Forward engineering-** direct mapping of a UML model into code.
- **Reverse engineering-** reconstruction of a UML model from an implementation.
- **Re-engineering-** Understanding existing software and modifying it.

UML- A language for…

4. Documenting
- Architecture
- Requirements
- Tests
- Activities
  - ✓ Project Planning
  - ✓ Release Management

3 basic building blocks of UML

Building Blocks of UML



3 basic building blocks of UML

- **Things**
  important modeling concepts
- **Relationships**
  tying individual things together
- **Diagrams**
  grouping interrelated collections of thin

Why model?

- **Analyse** the **problem-domain**
  - simplify reality
  - capture requirements
  - visualize the system in its entirety
  - specify the structure and/or behaviour of the system
- **Design** the **solution**
  - document the solution - in terms of its structure, behavior, etc.

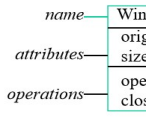
Conceptual Model of UML

- A conceptual model needs to be formed by an individual to understand UML.
- **UML contains three types of building blocks: things, relationships, and diagrams.**
- **Things**
  - Structural things
    - ✓Classes, interfaces, collaborations, use cases, components, and nodes.
  - Behavioral things
    - ✓Messages and states.
  - Grouping things
    - ✓Packages
  - Annotational things
    - ✓Notes
- **Relationships**: dependency, association, generalization ,composition ,link ,aggregation etc..
- **Diagrams**: class, object, use case, sequence, collaboration, statechart, activity, component and deployment.

I. Structural Things- *7 Things*

1. Class

A description of a set of objects that share the same attributes, operations, relationships, and semantics.



2. Interface

A collection of operations that specify a service (for a resource or an action) of a class or component. It describes the externally visible

I. Structural Things

3. Collaboration

- Define an interaction among two or more cl
- Define a society of roles and other element
- Provide cooperative behavior.
- Capture structural and behavioral dimensio
- UML uses 'pattern" as a synonym

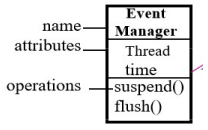
I. Structural Things

4. Use Case

- A sequence of actions that produce an obser result for a specific actor.
- A set of scenarios tied together by a commor goal.
- Provides a structure for behavioral things.
- Realized through a collaboration

I. Structural Things

5. Active Class



- Special class whose objects own one or m *processes* or *threads*.

I. Structural Things

6. Component

- Replaceable part of a system.
- Components can be packaged logically.
- Conforms to a set of interfaces.
- Provides the realization of an interface.
- Represents a physical module of code

I. Structural Things

7. Node

- Element that exists at *run time*.
- Represents a *computational resource*.
- Generally has memory and processing pov

II. Behavioral Things

- **Verbs** of UML Model
- **Dynamic parts** of UML models- *behaviour over time and space.*
- Usually **connected** to **structural things** in UML.

II. Behavioral Things

Two primary kinds of behavioral things:

Interaction

behavior of a set of objects comprising of a set o exchanges within a particular context to acc specific purpose.

State Machine

behavior that specifies the sequences of states a an interaction goes through during its lifetime in r events, together with its responses to those events

III. Grouping Things

Packages –

- one primary kind of grouping.

- General purpose mechanism for organizing eleme groups.
- Purely conceptual; only exists at development time.
- Contains behavioral and structural things.
- Can be nested.

IV. Annotational Things

- Explanatory parts of UML models
- Comments regarding other UML element called adornments in UML)

Note is the one primary annotational thing in UM

- best expressed in informal or formal text.

Conceptual Model- Relationship

- Dependency
- Association
- Generalization

Conceptual Model - Diagrams

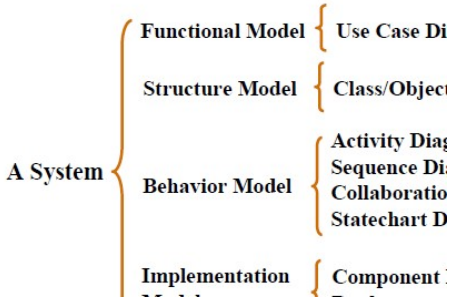
•Graphical representation of a set of elements.

•Represented by a connected graph:
- •Vertices are things;
- •Arcs are behaviors.

•5 most common views built from 9 diagram type

1. **Class Diagram; Object Diagram**
2. **Use case Diagram**
3. **Sequence Diagram; Collaboration D**
4. **Statechart Diagram**
5. **Activity Diagram**

Different perspectives of a system



Architectural Views and Diagrams

- **User model view**
  - relies on ***use case diagrams*** to describe the problem and its solution from the perspective of the customer or end user of a product
- **Structural model view**
  - describes static aspects of the system through ***class diagrams*** and ***object diagrams***
- **Behavioral model view**
  - specifies dynamic aspects of the system through ***sequence diagrams, collaboration diagrams, state diagrams, and activity diagrams***

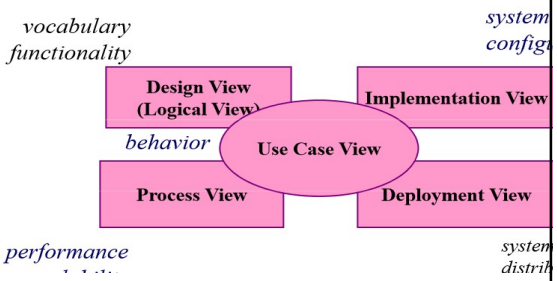
Architectural Views and Diagrams

- **Implementation model view**
  - concentrates on the **specific realization** of a solution, and depicts the organization of solution components in ***component diagrams***
- **Environment model view**
  - shows the **configuration of elements** in the environment, and indicates the mapping of solution components to those elements through **deployment diagrams**

4+1 Architecture of UML

- Architecture refers to the **different perspectives** from which a **complex system** can be **viewed.**
- The architecture of a *software-intensive system* is best described by five interlocking views:
  - **Use case view**: system as seen by **users, analysts** and **testers**.
  - **Design view**: **classes, interfaces and collaborations** that make up the system.
  - **Process view**: **active classes** (threads).
  - **Implementation view**: **files** that include the **system**.
  - **Deployment view**: **nodes** on which **SW resides**.

4+1 Architecture of UML



UML Concepts- The 4+1 View

- Use Case view
  - ✓Understandability
- Logical View
  - ✓Functionality
- Process View
  - ✓Performance
  - ✓Scalable
  - ✓Throughput
- Implementation View
  - ✓Software management
- Deployment View
  - ✓System topology
  - ✓Delivery
  - ✓Installation

I. Use Case View(*User View*)

- Technique to capture business process fro perspective.
- Encompasses the behavior as seen by users and testers.
- Specifies forces that shape the architecture.
- Static aspects captured in use case diagrams.

II. Design View(*Logical View*)

- Encompasses classes, interfaces, and collabor that define the vocabulary of a system.
- Supports functional requirements of the system
- Static aspects captured in class diagrams and diagrams.
- Dynamic aspects captured in interaction, state

III. Process View

- Encompasses the threads and processes concurrency and synchronization.
- Addresses performance, scalability, and throu
- Static and dynamic aspects captured as in

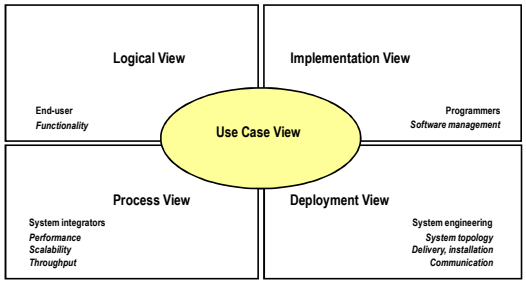
IV. Implementation View

- Encompasses components and files used to ass release a physical system.
- Addresses configuration management.
- Static aspects captured in component diagrams.
- Dynamic aspects captured in interaction, sta

V. Deployment View

- Encompasses the nodes that form the system topology.
- Addresses distribution, delivery, and installation.
- Static aspects captured in deployment diagrams.

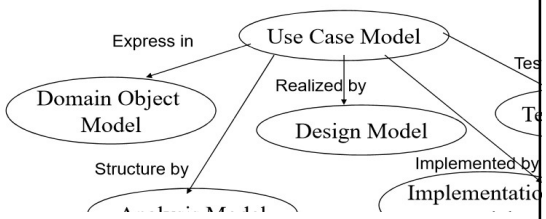
Representing System Architecture



UML Concepts in a nutshell

- Display the boundary of a system & functions using use cases and actors
- Illustrate use case realizations with diagrams
- Represent a static structure of a system u diagrams
- Model the behavior of objects with state diagrams
- Reveal the physical implementation a

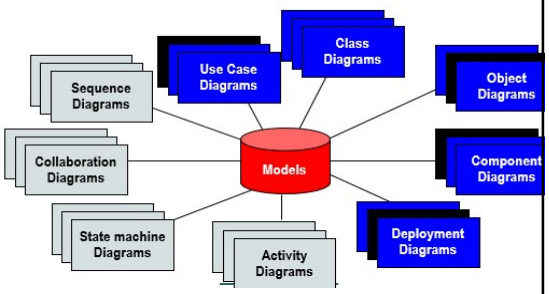
UML Concepts in a nutshell



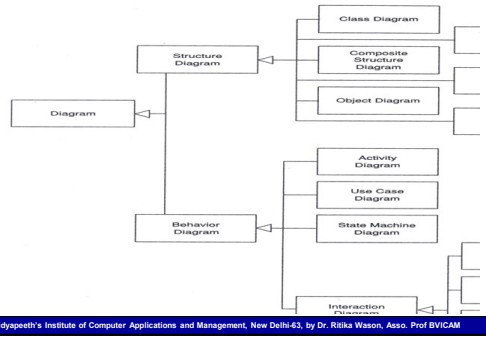
UML Diagram Classification

- **User model view**
  - use case diagrams
- **Structural model view**
  - class diagrams
  - object diagrams
- **Behavioral model view**
  - sequence diagrams
  - collaboration diagrams
  - state machine diagrams
  - activity diagrams
- **Implementation model view**
  - component diagrams
- **Environment model view**
  - deployment diagrams

UML Standardized Diagrams

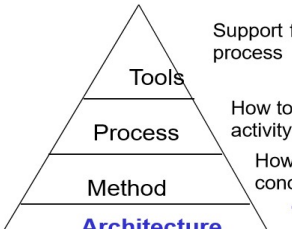


UML Diagram Classification



Architecture

System Architecture



Support for architecture, method process

How to scale up the method to activity

How to apply the concept(step by step)

The selected approac

System Architecture

- **System development** includes the development of **different models** of a software system
- **Aim** is to find **powerful modeling language**, **notation** or **modeling technique** for each model
- Set of **modeling techniques** defines **architecture** upon which the system development method is based
- The **architecture of a method** is the denotation of its set of modeling techniques

System Architecture

- **Modeling technique** is described by means of syntax, semantics and pragmatics
  - **Syntax** (How it looks)
  - **Semantics** (What it means)
  - **Pragmatics** (rules of thumb for using modeling technique)

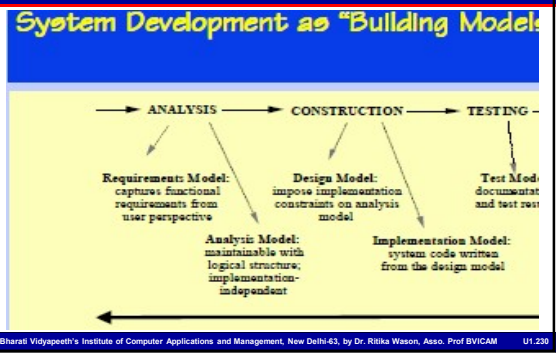
System Development

- **System development** is the work that occurs when we develop **computer support** to aid an **organizational procedures.**
- *System development is model building.*
- Commences with **identification** of **requirements**.
- **Specification** can be used for **contract** and to **plan** and **control development process**.
- Complex processes are often handled poorly. **OOSE** steps in from start to end of system life cycle.

Objectory Models

- Discipline process for the industrialized **development of software,** based on a **use-case driven design**
- **Built around several different models**
  - Requirement Model
  - Domain object model
  - Analysis Model
  - Design Model
  - Implementation model
  - Test model

System development is *Model Building*



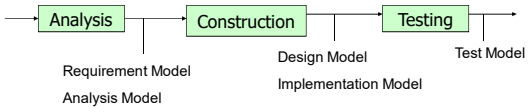
Model Architecture

Five different models

- The requirement model
  - ✓**Aims to capture the functional requirements**
- Analysis model
  - ✓**Give the system a strong and changeable object structure**
- Design model
  - ✓**Adopt and refine the object structure to the current implementation environment**

Model Architecture

- Implementation model
  - **Implement the system designed so far**
- Test model
  - **Verify whether the right system has been built or not**



Model Architecture

The Analysis Model

• Consisting of **various object classes: control object, entity objects,** and **interface objects.**

•The purpose of this model is to find a **robust** and **extensible structure** for the system as a **base for construction.**

• Each of the **object types** has its own special purpose for this robustness, and together they will offer the **total functionality** that was specified in the **Requirements Model**.

Model Architecture

The Construction Model

- We build our system through **construction** based on the **Analysis Model** and the **Requirements Model** created by the **analysis process**.
- The construction process lasts until the **coding is completed** and the included **units** have been tested.
- There are three main reasons for a construction process:

1) **The Analysis Model** is **not sufficiently formal**.
2) **Adaptation** must be made to the **actual implementation environment.**
3) We want to do **internal validation** of the **analysis results.**

Model Architecture

•The construction activity produces **two models,** the **Design Model** and the **Implementation Model.**

• Construction is thus divided into **two phases**; ***design and implementation***, each of which develops a model.

- ✓ The **Design Model** is a further **refinement** and **formalization** of the **Analysis Model** where consequences of the **implementation environment** have been taken into account.
- ✓ The **Implementation model** is the **actual implementation (code)** of the system**.**

Model Architecture

The Testing Model

• Testing is an activity to **verify** that a **correct system is being built.**

• Testing is **traditionally** an expensive activity, primarily because many ***faults are not detected*** until late in the **development**.

• To do **effective testing** we must have as a **goal** that *every test should detect a fault*

Development Processes

- Instead of focusing on how a **specific project** should be **driven**, the focus of the process is on how a certain product should be *developed* and *maintained* during its life cycle
- Divide the development work for a specific product into **processes**, where each of the processes describes *one activity of the management of a product.*
- **Processes** works in a highly **interactively** manner.
- Process handles the **specific activity** of the system development

Development Processes

- Architecture forms the **basis** of the method and process, that is the concept of each model
- Development can be regard as a set of **communicating processes**
- **System development** depends on these processes

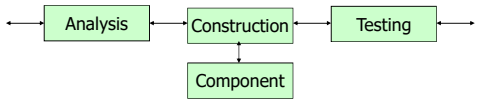
Development Processes

- All **development work** is managed by these processes.
- Each **process** consist of a **number of communicating sub processes.**
- **Main processes are**
  - Analysis
  - Construction
    - ✓**Component**
  - Testing

Processes and Models

- **Models** of the system created during **development**
- To **design models process description** is required
- Each process takes *one or several models* and transform it into other models
- Final model should be **complete** and **tested**, generally consists of source code and documentation
- **System development** is basically concerned with developing models of the system

Development Processes



I. Analysis Process

- Creates **conceptual picture**
- **Output** are **requirement model** and **Analysis model**
- Requirement Model
  - ✓**Done by use cases in the use case model**
  - ✓**Form the basis of construction and testing process**
  - ✓**Forms the basis of analysis Model**

Development Processes

- ✓**Basis of system structure**
- ✓**Specify all the logical objects to be included in the system and how these are related and grouped**
- ✓**Provide input for the construction process**

II. Construction Process

- Develops **design model** and **implementation model**
- Includes the implementation and results in complete system
- **Design Model**
  - ✓**Each object will be fully specified**
  - ✓**Consider the implementation constraints**

Development Processes

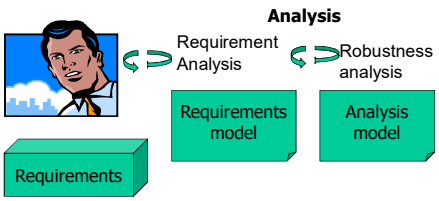
III. Testing Process

- **Integrates** the **system**, **verifies** it and decides whether it should be **delivered**

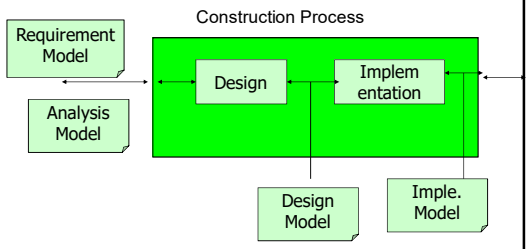
IV. Component development process

- Communicates with the **construction process**
- **Develops** and **maintain components**

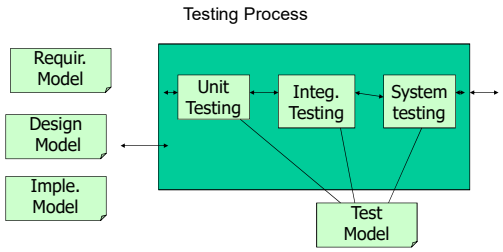
Development Processes - Analysis



Development Processes- Construction



Development Processes- Testing



OOSE Analysis Models

- Object-oriented software engineering (OOSE) proposes two analysis models for understanding the **problem domain**
  - Requirements Model
  - Analysis Model

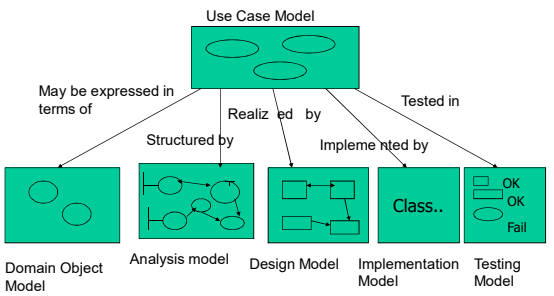
The **requirements model** serves two main **purposes**
- To delimit the **system**
- To define the **system functionality**

Requirement Model

- **Conceptual model** of the system is developed using:
  - **Problem domain objects**
  - **Specific interface descriptions** of the system (if meaningful to the system being developed)

⊗ The system is described as a *number of use cases* that are performed by a *number of actors*
- **Actors** constitute the **entities** in the **environment** of the system
- **Use cases** describe what takes places **within the system**
- A use case is a **specific way** of **using** the system by performing **some part** of the system functionality

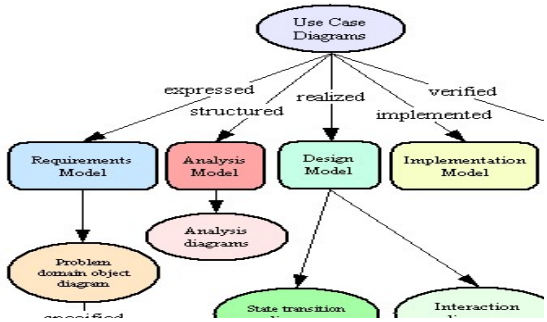
Requirement Model



Requirement Model

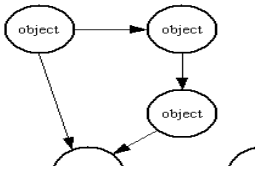
- The requirements model for the will comprise three main models of representation:
  - The use case model
  - The problem domain model
  - User interface descriptions

Requirement Model



Problem Domain Object Model

Provides a logical view of the system, which is used to specify the use cases for use case diagrams



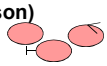
Object Oriented Analysis

- **Identifying objects**: Using concepts, CRC cards, stereotypes, etc.
- **Organising the objects**: classifying the objects identified, so similar objects can later be defined in the same class.
- **Identifying relationships between objects**: this helps to determine inputs and outputs of an object.
- **Defining operations of the objects**: the way of processing data within an object.
- **Defining objects internally**: information held within the objects.

Object Oriented Analysis Approaches

1.Analysis model with stereotypes (Jacobson)

- **Boundaries, entities, control.**

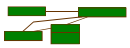


2.CRC cards (Beck, Cunningham)

- **Index cards and role playing.**

3.Conceptual model (Larman)

- **Produce a "light" class diagram.**



A good analyst knows more than one strategy and even may mix strategies

The Analysis Phase

- Begins with a **problem statements** generated during **system conception.**
- In software engineering, analysis is the process of **converting** the user requirements to system specification (system means the software to be developed).
- System specification, also known as the **logic structure**, is the developer's view of the system.
- **Function-oriented analysis**
  - Concentrating on the **decomposition** of complex functions to simply ones.
- **Object-oriented analysis**
  - Identifying **objects** and the **relationship** between objects.

Analysis Model

⊗ The analysis model gives a **conceptual configuration** of the system.

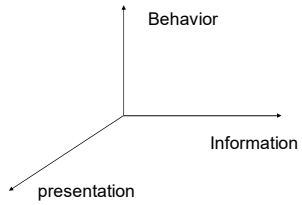
It consists of:

- The entity objects
- Control objects
- Interface objects

⊗ The analysis model forms the **initial transition** to **object-oriented design**

Dimensions of Analysis Model

Dimensions of the analysis model



Analysis Model- *Objects*

Entity object

- Information about an entity object is stored even after a use case is completed.

Control object

- A control object shows functionality that is not contained in any other object in the system

Interface object

- Interface objects interact directly with the environment

Requirement Model Structured in Analysis Model

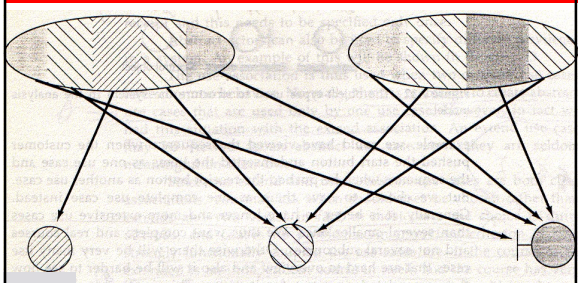


Figure 7.13 Each use case is distributed among analysis objects. Several use cases can have objects in common.

Design Model

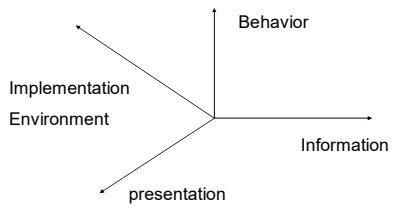
- Developed based on the analysis model
  - Implementation environment is taken into consideration
- The considered **environment factors** includes
  - Platform
  - Language
  - DBMS
  - Constraints
  - Reusable Components
  - Libraries
  - so on..

Design Model

- Design objects are different from analysis objects
- **Models**
  - Design object interactions
  - Design object interface
  - Design object semantics
    - ✓**(i.e., algorithms of design objects' operations)**
- More closer to the **actual source code**

Design Model Dimensions

Dimensions of the Design model



Design Model

- Use **block term** in place of object
- Sent from one block to another to trigger an execution
- A typical **block** is **mapped** to **one file**
- To manage system **abstractly** subsystem concept is introduced
- **Analysis Model** is viewed as **conceptual** and **logical model,** whereas the **design model** should take as closer to the **actual source code**
- Consist of explained source code
- OO language is desirable since all fundamentals concepts can easily be mapped onto **language constructs**
- Strongly desirable to have an easy match between a **block** and the **actual code module**

Implementation Model

- Consists of **annotated source code.**
- Object oriented language is desirable since all **fundamental concepts** can be easily **mapped** onto **language constructs**.
- Strongly desirable to have an easy **match** between a **block** and the **actual code module.**

Test Model

Fundamental concepts are **test specifications** and the **test results**

References

1. Ivar Jacobson, "Object Oriented Software Engineering", Pearson, 2004.
2. Grady Booch, James Runbaugh, Ivar Jacobson, "The UML User Guide", Pearson, 2004
3. R. Fairley, "Software Engineering Concepts", Tata McGraw Hill, 1997.
4. P. Jalote, "An Integrated approach to Software Engineering", Narosa, 1991.
5. Stephen R. Schach, "Classical & Object Oriented Software Engineering", IRWIN, 1996.
6. James Peter, W Pedrycz, "Software Engineering", John Wiley & Sons
7. Sommerville, "Software Engineering", Addison Wesley, 1999.
8. http://www.gentleware.com/fileadmin/media/archives/userguides/poseidon_users_guide/userguide.html
9. http://www.gentleware.com/fileadmin/media/archives/userguides/poseidon_users_guide/statemachinediagram.html
10. http://www.developer.com/design/article.php/2238131/State-Diagram-in-UML.htm
11. http://www.mariosalexandrou.com/methodologies/extreme-programming.asp

OBJECT ORIENTED ANALYSIS and DESIGN

Project Management & Inception & Analysis UNIT II

Learning Objectives

Project Management & Inception Phase

Analysis

- Introduction
- The requirements model
- The analysis model

UML: Use case Diagram, Class Diagram, Object Diagram, Activity Diagram, Sequence Diagram

Project Management & Inception phase

A lot of computer science people think:

"I don't want to code for the rest of my life. Maybe I would enjoy managing the project."

What do you think are some of the tasks you would be doing if you were a project manager?

Would you still code?

Would you miss coding ☺?

Is it important the project manager be able to code?

What is Project Management?

Project management encompasses all the activities needed to plan and execute a project:

- Deciding what needs to be done
- Estimating costs
- Ensuring there are suitable people to undertake the project
- Defining responsibilities
- Scheduling
- Making arrangements for the work
- *continued ...*

What is Project Management?

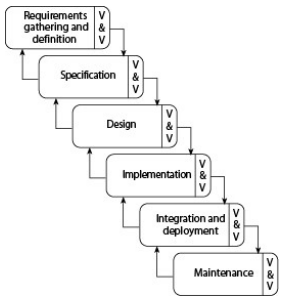
- Directing
- Being a technical leader
- Reviewing and approving decisions made by others
- Building morale and supporting staff
- Monitoring and controlling
- Co-ordinating the work with managers of other projects
- Reporting
- Continually striving to improve the process

Software Process Models

Software process models are general approaches for organizing a project into activities.

- Help the project manager and his or her team to decide:
  - ✓What work should be done;
  - ✓In what sequence to perform the work.
- The models should be seen as *aids to thinking*, not rigid prescriptions of the way to do things.
- Each project ends up with its own unique plan.

The waterfall model



The waterfall model

The classic way of looking at S.E. that accounts for the importance of requirements, design and quality assurance.

- The model suggests that software engineers should work in a series of stages.
- Before completing each stage, they should perform quality assurance (verification and validation).
- The waterfall model also recognizes, to a limited extent, that you sometimes have to step back to earlier stages.
- **QUESTION: What is wrong with getting all the requirements completed upfront (like I have done for you with our project)?**

Limitations of the waterfall model

- The model implies that you should attempt to complete a given stage before moving on to the next stage
  - ✓Does not account for the fact that requirements constantly change.
  - ✓It also means that customers can not use anything until the entire system is complete.
- The model makes no allowances for prototyping.
- It implies that you can get the requirements right by simply writing them down and reviewing them.
- The model implies that once the product is finished, everything else is maintenance.

Reengineering

Periodically project managers should set aside some time to re-engineer part or all of the system

- The extent of this work can vary considerably:
  - ✓Cleaning up the code to make it more readable.
  - ✓Completely replacing a layer.
  - ✓*Re-factoring* part of the design.
- In general, the objective of a re-engineering activity is to increase maintainability.

Cost estimation

To estimate how much software-engineering time will be required to do some work.

- *Elapsed time*
  - ✓The difference in time from the start date to the end date of a task or project.
- *Development effort*
  - ✓The amount of labour used in *person-months* or *person-days*.
  - ✓To convert an estimate of development effort to an amount of money:
    - You multiply it by the *weighted average cost* (*burdened* cost) of employing a software engineer for a month (or a day).

Question

Assume that I gave your group the task of figuring out how much time you needed to code your project because you were going sell it online.

What are some techniques/ideas/concerns/thoughts you have for estimating the timing of a large project?

i.e. How do you decide/figure out how long it takes you to do an assignment?

Principles of effective cost estimation

Principle 1: Divide and conquer.

- To make a better estimate, you should divide the project up into individual subsystems.
- Then divide each subsystem further into the activities that will be required to develop it.
- Next, you make a series of detailed estimates for each individual activity.
- And sum the results to arrive at the grand total estimate for the project.

Principles of effective cost estimation

Principle 2: Include all activities when making estimates.

- The time required for *all* development activities must be taken into account.
- Including:
  - Prototyping
  - Design
  - Inspecting
  - Testing
  - Debugging
  - Writing user documentation
  - Deployment.

Principles of effective cost estimation

Principle 3: Base your estimates on past experience combined with knowledge of the current project.

- If you are developing a project that has many similarities with a past project:
  - ✓ You can expect it to take a similar amount of work.
- Base your estimates on the *personal judgement* of your experts

  or
- Use *algorithmic models* developed in the software industry as a whole by analyzing a wide range of projects.
  - ✓They take into account various aspects of a project's size and complexity, and provide formulas to compute anticipated cost.

Algorithmic models

Allow you to systematically estimate development effort.

- Based on an estimate of some other factor that you can measure, or that is easier to estimate:
  - ✓The number of use cases
  - ✓The number of distinct requirements
  - ✓The number of classes in the domain model
  - ✓The number of widgets in the prototype user interface
  - ✓An estimate of the number of lines of code

Algorithmic models

- A typical algorithmic model uses a formula like the following:
  - ✓COCOMO:

$$E = a + bN^c$$

  - ✓Functions Points:

$$S = W_1F_1 + W_2F_2 + W_3F_3 + \ldots$$

Principles of effective cost estimation

Principle 4: Be sure to account for *differences* when extrapolating from other projects.

- Different software developers
- Different development processes and maturity levels
- Different types of customers and users
- Different schedule demands
- Different technology
- Different technical complexity of the requirements
- Different domains
- Different levels of requirement stability

Principles of effective cost estimation

Principle 5: Anticipate the worst case and plan for contingencies.

- Develop the most critical use cases first
  - ✓If the project runs into difficulty, then the critical features are more likely to have been completed
- Make three estimates:
  - ✓Optimistic (O)
    - Imagining a everything going perfectly
  - ✓Likely (L)
    - Allowing for typical things going wrong
  - ✓Pessimistic
    - Accounting for everything that could go wrong

Principles of effective cost estimation

Principle 6: Combine multiple independent estimates.

- Use several different techniques and compare the results.
- If there are discrepancies, analyze your calculations to discover what factors causing the differences.
- Use the Delphi technique.
  - ✓Several individuals initially make cost estimates in private.
  - ✓They then share their estimates to discover the discrepancies.
  - ✓Each individual repeatedly adjusts his or her estimates until a consensus is reached.

Principles of effective cost estimation

Principle 7: Revise and refine estimates as work progresses

- As you add detail.
- As the requirements change.
- As the risk management process uncovers problems.

Skills needed on a team

- Architect
- Project manager
- Configuration management and build specialist
- User interface specialist
- Technology specialist
- Hardware and third-party software specialist
- User documentation specialist
- Tester

Project Scheduling and Tracking

- *Scheduling* is the process of deciding:
  - ✓In what sequence a set of activities will be performed.
  - ✓When they should start and be completed.
- *Tracking* is the process of determining how well you are sticking to the cost estimate and schedule.

Some Basic Project Management Terminology

- *Deliverable:* some concrete thing which is to be delivered, to the client or internally to the development team; e.g.
  - Specifications reports
  - Executable program
  - Source code
- *Task/Activity*: something we have to do during the project; e.g.
  - Defining user requirements
  - Coding a module
  - Doing system testing
- Each task or activity will take some length of time
  - Referred to as ***duration*** of task
  - Sometimes measured in days, weeks, etc.
  - Sometimes measured in person-days, person-weeks, etc.
  - ***Person-day*** = number of people X number of days
    - ✓ Example: 12 person days for writing all code could mean 1 person 12 days or 4 people 3 days
    - ✓ **Note**: not always true that a task that takes 1 programmer 12 days would take 12 programmers 1 day

Dependencies and Milestones

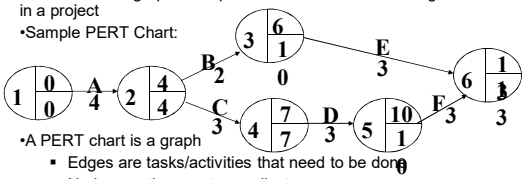
- For a given task or activity, may be impossible to start it without some other task(s) or activity(ies) having been completed; e.g.
  - Cannot start coding without completing design
  - Cannot start system testing without completing code integration and test plan
- If task B cannot start without A being completed, we say
  - B depends on A
  - There is a dependency between A and B
- *Milestone*: some achievement which must be made during the project; e.g.
  - Delivering some deliverable
  - Completing some task
- Note, delivering a deliverable may be a milestone, but not all milestones are associated with deliverables

Setting and Making Deadlines

- *Deadline* time by which milestone has to be met
  - Some deadlines are set by the client
  - Others are set by us on project to make sure project stays on track
- To set a deadline for completing task T, we must consider how long it will take to:
  - Complete the tasks that task T depends on
  - Complete task T itself
- If we miss a deadline, we say (euphemistically) "the deadline has slipped"
  - This is virtually inevitable
- Important tasks for project managers
  - Monitor whether past deadlines have slipped
  - Monitor whether future deadlines are going to slip
  - Allocate or reallocate resources to help make deadlines
- PERT chart and Gantt charts help project managers do these things (among others)

PERT Charts

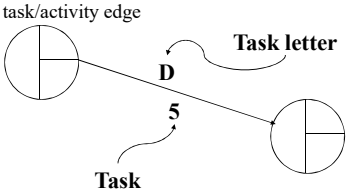
- PERT = Project Evaluation and Review Technique
- PERT chart = graphical representation of the scheduling of events in a project
- Sample PERT Chart:



- A PERT chart is a graph
  - Edges are tasks/activities that need to be done
  - Nodes are the events or milestones
- Task edge T from event node E1 to event node E2 signifies:
  - Until event E1 happens, task T cannot be started
  - Until task T finishes, event E2 cannot happen
- Events often simply represent completion of tasks associated with arrows entering it

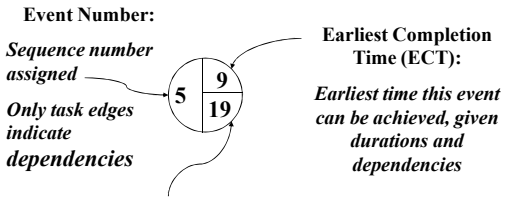
PERT Chart Task Edges

- Parts of a task/activity edge



- Task letter:
  - Often keyed to a legend to tell which task it represents
- Task duration = how long (e.g. days, hours) task will take

PERT Chart Event Nodes



Building a PERT Chart

Steps:

1. Make a list of all project tasks (and events if possible).
2. Find interrelated task dependencies (what task has to be completed before other tasks)
3. Draw initial PERT without durations, ECTs or LCTs
4. Estimate duration of each task
5. Fill in durations
6. Calculate ECTs and LCTs

•We will do this for an example system:

→ Generic software system with 3 modules

Example: Generic Software Project

TASK ID	Task Description
A	Specification
B	High Level Design
C	Detailed Design
D	Code/Test Main module
E	Code/Test DB module
F	Code/Test UI module
G	Write test plan
H	Integrate/System Test
I	Write User Manual
J	Typeset User Manual

• To start PERT chart: identify dependencies between tasks

Dummy Tasks

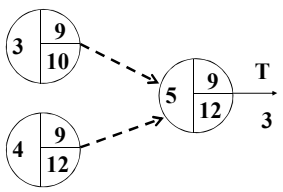
Sometimes it is necessary to use *dummy tasks*:

- Shows the dependency between 2 events where no activity is performed

Example:

- Events 3, 4 signify the compilation of separate modules.
- Create an event 5 to signify "all modules compiled together".

Denote dummy tasks using dash lines

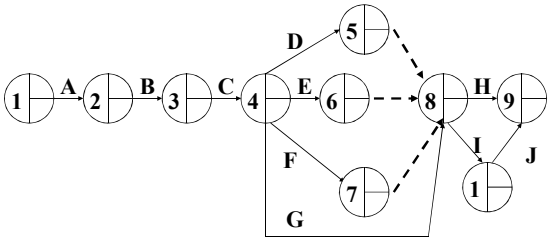


Example: Tasks with Dependencies

To start the PERT, identify the dependencies amongst tasks

TASK ID	Task Description	Preceed ID	Succ. ID
A	Specification	1	2
B	High Level Design	2	3
C	Detailed Design	3	4
D	Code/Test Main	4	5
E	Code/Test DB	4	6
F	Code/Test UI	4	7
G	Write test plan	4	8
	Dummy Task	5	8
	Dummy Task	6	8
	Dummy Task	7	8
H	Integrate/System Test	8	9
I	Write User Manual	8	10
J	Typeset User Manual	10	9

Software Example: Skeleton PERT Chart



Note: dummy tasks connecting events 5, 6 and 7 to 8

Estimating Durations

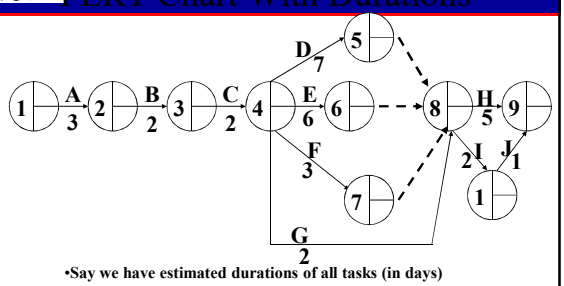
Suggestions for estimating durations of tasks:

- Don't just make up a number
- Look at previous similar tasks from other projects and use those as guidelines
- Try to identify factors such as difficulty, skill level
  - ✓ Each weighting factor will help you make a better estimate

Factors to consider:

- Difficulty of task
- Size of team
- Experience of team
- Number, attitude and availability of end users
- Management commitment
- Other projects in progress

PERT Chart With Durations



- Say we have estimated durations of all tasks (in days)
- New PERT chart, with durations filled in:
- Note, dummy tasks (dashed lines) always have a duration of zero

Calculating ECTs

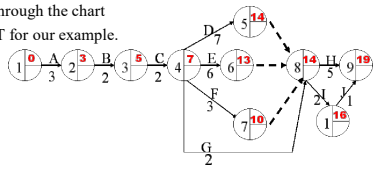
ECT = earliest time event can be completed

To calculate:

- For an event not depending on others: ECT = 0
  - ✓ Usually this is the first event
- For an event E depending on one or more others:
  - ✓ Calculate ECTs of event(s) that E depends on
  - ✓ Add duration(s) of task(s) leading to E
  - ✓ If E depends on more than one event, take MAX

Proceed left to right (→) through the chart

Exercise: calculate the ECT for our example.



Calculating LCT

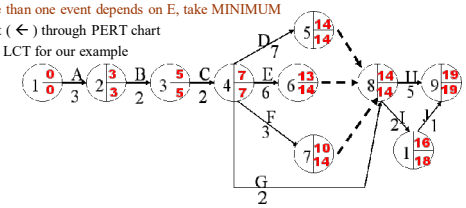
LCT = latest time event can be completed, while still finishing last ask at indicated time

To calculate:

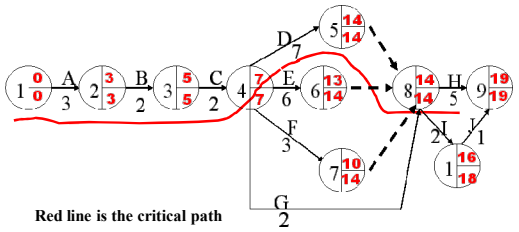
- For an event which no other events depend on: LCT = ECT
  - ✓ Generally there will only be one such event
- For an event E which one or more others depend on:
  - ✓ Calculate LCTs of event(s) that depend on E
  - ✓ Subtract duration(s) of task(s) leading from E
  - ✓ If more than one event depends on E, take MINIMUM

Proceed right to left (←) through PERT chart

Exercise: calculate LCT for our example



Critical Path



Red line is the critical path

What does it represent?

Uses of PERT Charts

We can use PERT charts for:

- Determining the estimated time to complete a project
- Deriving actual project dates
- Allocating resources
- Identifying potential and current problems (is one task behind schedule?, can we shuffle people?)

Critical Path: Path through chart such that if any deadline slips, the final deadline slips (where all events have ECT = LCT (usually there is only one)

In software example:

- Task I is not on the critical path: even if we don't finish it until time 18, we're still okay
- Task D is on the critical path: if we don't finish it until for example, time 16, then:
  - ✓ We can't start task H (duration 3) until time 16
  - ✓ So we can't complete task H until time 21

We can use PERT charts for

- Identifying the critical path
- Reallocating resources, e.g. from non-critical to critical tasks.

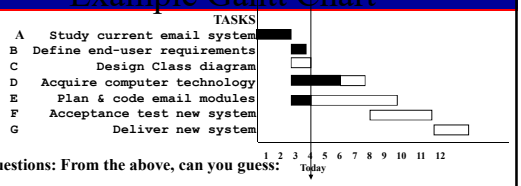
PERT Chart Exercise

Task	Prec Tasks	Description	Time(hrs)
A	none	decide on date for party	1
B	A	book bouncy castle	1
C	A	send invitations	4
D	C	receive replies	7
E	D	buy toys and balloons	1
F	D	buy food	3
G	E	blow up balloons	2
H	F	make food	1
I	H, G	decorate	1
J	B	get bouncy castle	1
K	J, I	have party	1
L	K	clean up	4
M	K	send back bouncy castle	1
N	L	send thank you letters	3
O	M	donate unwanted gifts	3

Gantt Charts

- Graphical Representation of a schedule
- Helps to plan, coordinate and track specific tasks in a project
- Named after Henry Gantt who invented them in 1917
- Depicts some of the same information as on a PERT chart
- Also depicts new information

Example Gantt Chart



Questions: From the above, can you guess:

- **Which, if any, tasks should have been completed by today and aren't even started? _____**
- **Which, if any, tasks have been completed? _____**
- **Which, if any, tasks have been completed ahead of schedule:? _____**
- **Which, if any, tasks are on or ahead of schedule? _____**
- **Which, if any, tasks are behind schedule? _____**

Building and Using a Gantt Chart

Steps for building a Gantt Chart

1. Identify the tasks to be scheduled
2. Determine the durations of each task
3. List each task down the vertical axis of chart
    1. In general, list tasks to be performed first at the top and then move downward as the tasks will happen
4. Use horizontal axis for the dates
5. Determine start and finish dates for activities
    1. Consider which tasks must be completed or partially completed before the next task

To use the Gantt chart to report progress:

- If the task has been completed, completely shade in the bar corresponding to the task
- If the task has been partially completed, shade in the percentage of the bar that represents the percentage of the task that has been completed
- Unshaded bars represents tasks that have not been started.

Gantt Chart: Exercise

Task	Prec Tasks	Description	Time(hrs)
A	none	decide on date for party	1
B	A	book bouncy castle	1
C	A	send invitations	4
D	C	receive replies	7
E	D	buy toys and balloons	1
F	D	buy food	3
G	E	blow up balloons	2
H	F	make food	1
I	H, G	decorate	1
J	B	get bouncy castle	1
K	J, I	have party	1
L	K	clean up	4
M	K	send back bouncy castle	1
N	L	send thank you letters	3
O	M	donate unwanted gifts	3

Gantt Chart: Exercise

Draw the Gantt chart using the following criteria:

- label hours 0 to 30 across the horizontal axis
- Mark a review stage at hour 14 to monitor the progress
- Assume and illustrate that tasks A, B, C and D have been completed at hour 14
- State which tasks are ahead and which tasks are behind schedule
- NOTE: if you are using MS Project and want a different unit of time, just type 2 hours (instead of 2 days). ALSO, if you want to have a milestone, like Handing in Group Assignment 1, then give it a ZERO duration.

PERT vs. Gantt

PERT chart

- All us to show dependencies explicitly
- Allow us to calculate critical path
- Can tell us how one task falling behind affects other tasks

Gantt charts

- Allow us to record progress of project
- Allow us to see what tasks are falling behind
- Allow us to represent overlapping tasks

Project Management Tools, e.g. MS Project

- Allow us to specify tasks, dependencies, etc
- Allow us to specify progress on tasks, etc
- Can generate either PERT or Gantt charts (whichever we want) from data entered

Difficulties and Risks in Project Management

- **Accurately estimating costs is a constant challenge**
  - ✓ *Follow the cost estimation guidelines.*
- **It is very difficult to measure progress and meet deadlines**
  - ✓ *Improve your cost estimation skills so as to account for the kinds of problems that may occur.*
  - ✓ *Develop a closer relationship with other members of the team.*
  - ✓ *Be realistic in initial requirements gathering, and follow an iterative approach.*
  - ✓ *Use earned value charts to monitor progress.*

Difficulties and Risks in Project Management

- **It is difficult to deal with lack of human resources or technology needed to successfully run a project**
  - ✓ *When determining the requirements and the project plan, take into consideration the resources available.*
  - ✓ *If you cannot find skilled people or suitable technology then you must limit the scope of your project.*

Difficulties and Risks in Project Management

- **Communicating effectively in a large project is hard**
  - ✓ *Take courses in communication, both written and oral.*
  - ✓ *Learn how to run effective meetings.*
  - ✓ *Review what information everybody should have, and make sure they have it.*
  - ✓ *Make sure that project information is readily available.*
  - ✓ *Use 'groupware' technology to help people exchange the information they need to know*

Difficulties and Risks in Project Management

- **It is hard to obtain agreement and commitment from others**
  - ✓ *Take courses in negotiating skills and leadership.*
  - ✓ *Ensure that everybody understands*
    - *The position of everybody else.*
    - *The costs and benefits of each alternative.*
    - *The rationale behind any compromises.*
  - ✓ *Ensure that everybody's proposed responsibility is clearly expressed.*
  - ✓ *Listen to everybody's opinion, but take assertive action, when needed, to ensure progress occurs.*

Review

Draw a PERT Chart for the following activities:

Activity	Description	Predecessor	Estimated Time
A	Drive home	None	0.5
B	Wash Clothes	A	4.0
C	Pack	B	0.5
D	Go to bank	A	1.0
E	Pay bill	D	0.5
F	Pack car	C,E	0.5
G	Drive to bus	F	0.5

OOSE Analysis Models

- Object-oriented software engineering (OOSE) proposes two analysis models for understanding the **problem domain**
  - Requirements Model
  - Analysis Model

The **requirements model** serves two main **purposes**
- To delimit the **system**
- To define the **system functionality**

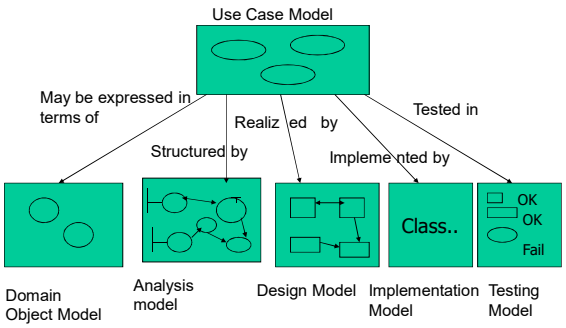
Requirement Model

- **Conceptual model** of the system is developed using:
  - **Problem domain objects**
  - **Specific interface descriptions** of the system (if meaningful to the system being developed)

⊗ The system is described as a *number of use cases* that are performed by a *number of actors*

- **Actors** constitute the **entities** in the **environment** of the system
- **Use cases** describe what takes places **within the system**
- A use case is a **specific way** of **using** the system by performing **some part** of the system functionality

Requirement Model

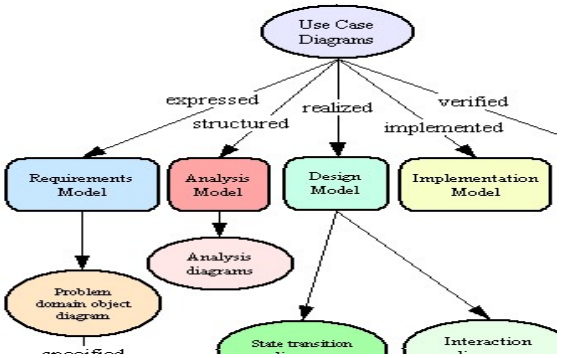


Requirement Model

❑ The requirements model for the will comprise three main models of representation:

- The use case model
- The problem domain model
- User interface descriptions

Requirement Model



Actors

- In order to identify use cases to be performed in the system, we need to first **identify system users**
- The **system users** are referred to as **actors**.
- Actors model the **prospective** '***users***' of the system.
- An actor is a **user type** or **category**. When an actor does something, the actor acts as an **occurrence** of that **type**.
- An actor may represent a **person** or **another system interacting** with the **intended system**

Actors and Role Play

- One person can **instantiate** (play the roles of) several different actors
- Actors **define** the **roles** that **users** can play
- Actors model anything that needs to **exchange information** with the **system**.
- Actors can **model human users** but they can also model other systems communicating with the intended system

Identifying Actors

- Actors constitute anything **external** to the system
- Identifying all the relevant actors for a system may require several **iterations**
- **General guidelines** include the following:
  - Ask yourself why the system is been developed
  - Who are people the system is intended to help?
  - What other systems are likely to interface with new system?

Primary and Secondary Actors

- Actors who ***use the system directly*** (or in their daily work) are known as **Primary actors**
- Primary actors are **associated** with one or more of the **main tasks** of the system
- Primary actors **govern** the **system structure**. Thus when ***identifying use cases***, we first start with the primary actors
- Actors who are concerned with **supervising** and **maintaining** the system are called **secondary actors**
- The **distinction** between the primary and secondary actors has a *bearing on the system structuring*

Use Cases

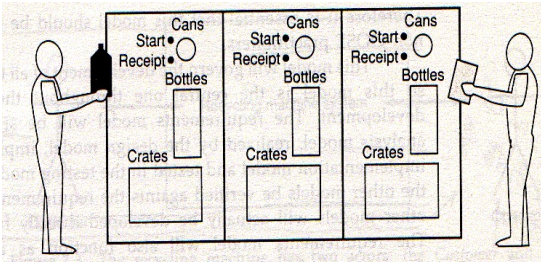
- After the actors have been identified the next step is to ***define the functionality of the system***. This is done by **specifying use cases.**
- Actors are a **major tool** in finding use cases. Each actor will perform a number of **use cases** in the system.
- Each use case constitutes a *complete course of events* initiated by an actor and specifies the **interaction** that takes place between the **actor** and the **system**
- A use case is a **special sequence** of **related transactions** performed by an **actor** and the **system** in dialogue.
- The *collective use cases* should specify all the **existing ways** of **using the system**

Recycling Machine Example

Throughout the discussion of the analysis and the construction activities, we will show how the different concepts are used in practice, by developing a system. The system controls a recycling machine for returnable bottles, cans and crates (used in Europe to hold several bottles). The machine can be used by several customers at the same time, and each customer can return all three types of items on the same occasion.

Since there may be different types and sizes of bottles and cans, the system has to check, for each item, what type was turned in. The system will register how many items each customer returns, and when the customer asks for a receipt, the system will print out what he turned in, the value of the returned items and the total return sum that will be paid to the customer.

Recycling Machine Example

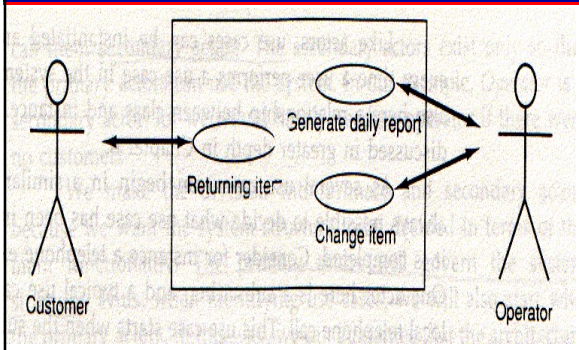


The recycling machine can receive several different types of returnable items from several customers at the same time.

Recycling Machine Example

The system is also used by an operator. The operator wants to know how many items of each type have been turned in during the day. At the end of the day, the operator asks for a printout of the total number of items that have been turned in to the system on that particular day. The operator should also be able to change information in the system, such as the deposit values of the items. If there is something amiss, for instance if a can gets stuck, or if the receipt roll is finished, the operator will be called by a special alarm signal.

Use Case Diagram



 U2.67

Use Case Description

Returning Item is started by *Customer* when he wants to return cans, bottles or crates. With each item that *Customer* places in the recycling machine, the system will increase the received number of items from *Customer* as well as the daily total of this particular type. When *Customer* has turned in all his items, he will press the receipt button to get a receipt where the returned items have been printed as well as a total return sum.

Generate Daily Report is started by *Operator* when he wants to print out information about the returned deposit items of the day. The system will print out how many of each deposit item type have been received this day, as well as the overall total for the day. The total number will be reset to zero to start a new daily report.

Change Item is used by *Operator* to change information in the system. The return value as well as the size of each returnable item can be changed, and new types of items can be added.

 U2.68

Case Study

Consider a **telephone exchange**:

- One actor is a subscriber and a typical use case to make a local telephone call. This use starts when the subscriber lifts his telephone receiver.
- Another use case is to order a wake-up call.
- Both use cases start when the subscriber lifts the telephone.
- However, when the subscriber lifts his telephone, its not obvious which use case he would like to perform.
- Thus uses cases may begin in a similar manner but we may not know which use case is to be carried out until its over.
- The actor should be viewed as someone who initiates a course of events that eventually results in a complete use case. Rather than someone who demands that a use case be performed.

 U2.69

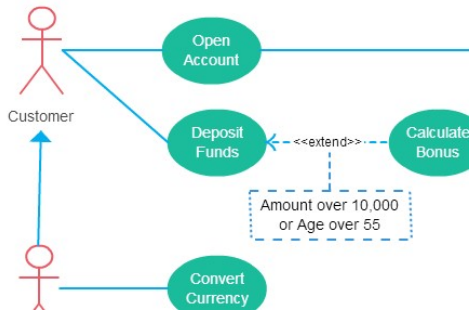
Use Case Relationships- Include

- **Use case include** is a directed relationship between two **use cases** which is used to show that behaviour of the **included** use case (the addition) is inserted into the **behaviour** of the **including** (the base) use case.
- The **include** relationship could be used:
  - To **simplify large use case** by splitting it into several use cases,
  - To **extract common parts** of the behaviours of two or more use cases.
- A large use case could have some behaviours which might be **detached** into distinct smaller use cases to be **included back** into the base use case using the UML **include** relationship.
- The **purpose** of this action is **modularization of behaviours**, making them more **manageable**.

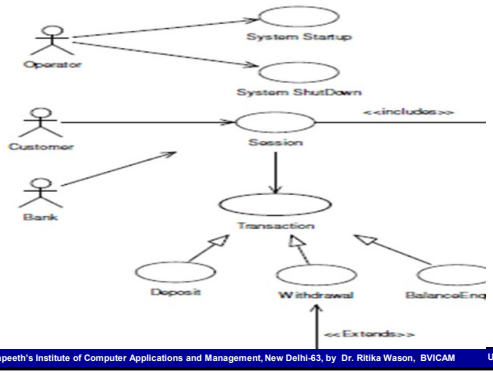
Use Case Relationships- Extend

- **Extend** is a **directed relationship** that specifies how and when the behaviour defined in usually supplementary (optional) **extending use case** can be inserted into the **behaviour** defined in the **extended use case**.
- **Extended** use case is meaningful on its own, it is **independent** of the extending use case.
- **Extending** use case typically defines **optional behaviour** that is not necessarily meaningful by itself.
- The extension takes place at one or more **extension points** defined in the **extended use case**.
- **Extend** relationship is shown as a dashed line with an open arrowhead directed from the **extending use case** to the **extended (base) use case**.

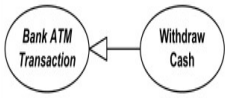
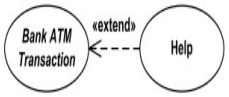
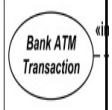
Use Case Relationships



Banking System Use Case Diagram



A Comparative Study

Generalization	Extend	
		
Base use case could be abstract use case (incomplete) or concrete (complete).	Base use case is complete (concrete) by itself, defined independently.	Base use case is incom
Specialized use case is required, not optional, if base use case is abstract.	Extending use case is optional, supplementary.	Included use case requ
No explicit location to use specialization.	Has at least one explicit extension location.	No explicit inclusion l

Requirement Engineering

Requirements

- Goal
  - To understand the problem
- Necessary to Understand Requirements
  - Organization
  - Existing Systems
  - Processes
  - Improvements
- Once you have all this information, now what?

Requirement Elicitation

- ✓Techniques
  - Interview / Meeting
  - Survey / Questionnaire
  - Observation
  - Ethnography / Temporary Assignment
  - Business Plans
  - Review Internal / External Documents
  - Review Software

Requirement Analysis

Requirements Analysis

- Goal
  - To bridge the gap between the problem domain and the technical domain
- Tasks
  - Problem Recognition
  - Evaluation and synthesis
  - Modeling
  - Specification
  - Review

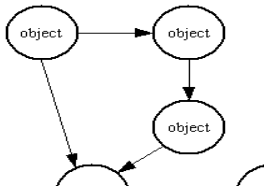
Requirement Analysis Principles

Requirements Analysis Principles

- Information domain of a problem must be represented and understood
- Models that depict system information, function, and behavior should be developed
- Models must be partitioned in a manner that uncovers detail in a layered fashion
- Analysis process should move from essential information toward implementation detail

Problem Domain Object Model

Provides a logical view of the system, which is used to specify the use cases for use case diagrams



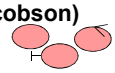
Object Oriented Analysis

- **Identifying objects**: Using concepts, CRC cards, stereotypes, etc.
- **Organising the objects**: classifying the objects identified, so similar objects can later be defined in the same class.
- **Identifying relationships between objects**: this helps to determine inputs and outputs of an object.
- **Defining operations of the objects**: the way of processing data within an object.
- **Defining objects internally**: information held within the objects.

Object Oriented Analysis Approaches

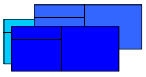
1.Analysis model with stereotypes (Jacobson)

- **Boundaries, entities, control.**



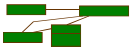
2.CRC cards (Beck, Cunningham)

- **Index cards and role playing.**



3.Conceptual model (Larman)

- **Produce a "light" class diagram.**



A good analyst knows more than one strategy and even may mix strategies

The Analysis Phase

- Begins with a **problem statements** generated during **system conception.**
- In software engineering, analysis is the process of **converting** the user requirements to system specification (system means the software to be developed).
- System specification, also known as the **logic structure**, is the developer's view of the system.
- **Function-oriented analysis**
  - Concentrating on the **decomposition** of complex functions to simply ones.
- **Object-oriented analysis**
  - Identifying **objects** and the **relationship** between objects.

Analysis Model

⊗ The analysis model gives a **conceptual configuration** of the system.

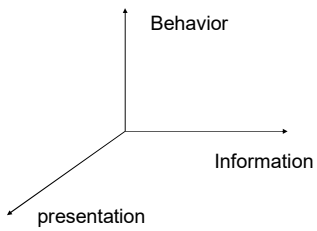
It consists of:

- The entity objects
- Control objects
- Interface objects

⊗ The analysis model forms the **initial transition** to **object-oriented design**

Dimensions of Analysis Model

Dimensions of the analysis model



Analysis Model- *Objects*

Entity object

- Information about an entity object is stored even after a use case is completed.

Control object

- A control object shows functionality that is not contained in any other object in the system

Interface object

- Interface objects interact directly with the environment

Requirement Model Structured in Analysis Model

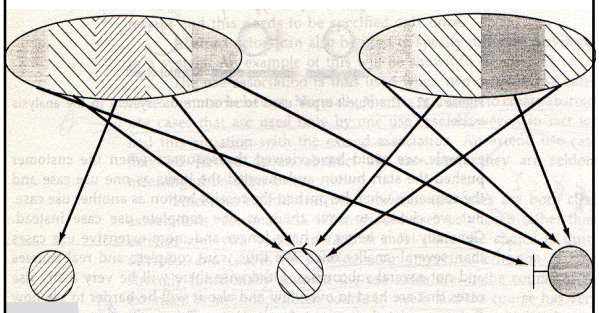


Figure 7.13 Each use case is distributed among analysis objects. Several use cases can have objects in common.

Design Model

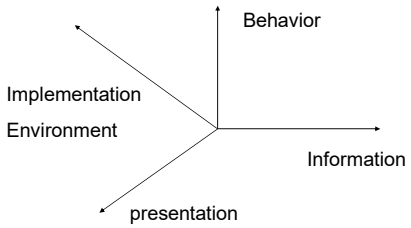
- Developed based on the analysis model
  - Implementation environment is taken into consideration
- The considered **environment factors** includes
  - Platform
  - Language
  - DBMS
  - Constraints
  - Reusable Components
  - Libraries
  - so on..

Design Model

- Design objects are different from analysis objects
- **Models**
  - Design object interactions
  - Design object interface
  - Design object semantics
    - ✓**(i.e., algorithms of design objects' operations)**
- More closer to the **actual source code**

Design Model Dimensions

Dimensions of the Design model



Design Model

- Use **block term** in place of object
- Sent from one block to another to trigger an execution
- A typical **block** is **mapped** to **one file**
- To manage system **abstractly** subsystem concept is introduced
- **Analysis Model** is viewed as **conceptual** and **logical model**, whereas the **design model** should take as closer to the **actual source code**
- Consist of explained source code
- OO language is desirable since all fundamentals concepts can easily be mapped onto **language constructs**
- Strongly desirable to have an easy match between a **block** and the **actual code module**

Implementation Model

- Consists of **annotated source code.**
- Object oriented language is desirable since all **fundamental concepts** can be easily **mapped** onto **language constructs**.
- Strongly desirable to have an easy **match** between a **block** and the **actual code module.**

Test Model

Fundamental concepts are **test specifications** and the **test results**

Analysis

Learning Objectives

- The Analysis Phase
- Analysis Model
- Meta Model of Analysis Model
- Analysis workflow details
- Analysis model-rules of thumb
- Object Oriented Analysis
- Three ways to do Object Oriented Analysis
- Conceptual Model – Overview
- The Concept Category List
- Finding Concepts with Noun Phrase Identification

Learning Objectives

- Exercise
- How to make a conceptual model
- Drawing of Concepts
- Adding Associations
- Adding Attributes
- The Object Oriented Analysis Model (Jacobson)
- Subsystem
- Good Analysis class
- Bad Analysis class

Analysis Model

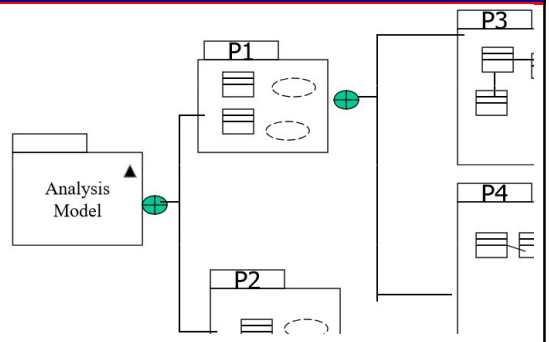
• The analysis model is intended to define a **first structure** of the system to be **designed** in the form of a **class** and/or an **object diagram**.

• The analysis model, gives the system its ***initial structure*** which is subject to further refinement in later development steps.

• According to the **Unified Process,** the development of the analysis model has to occur on the basis of the **use case specifications.**

• The analysis model has to be capable to fulfill the **functional requirements** stated in the **use case descriptions**.

Meta Model of Analysis Model

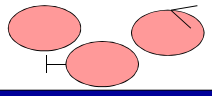


Analysis Model- Rule of Thumb

- 50 to 100 analysis classes in analysis model in an average system
- Include classes from the vocabulary of the problem domain
- Do not make implementation decisions
- Focus on classes and associations- minimize coupling
- Use inheritance whenever needed
- Keep it simple

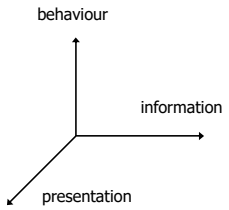
The Object Oriented Analysis Model (Jacobson)

- An **analysis model** is used to **represent** the **system specification.**
- To achieve **robustness** and **stability** the model must be **implementation environment independent.**
- Any change in the implementation environment will not affect the **logical structure** of the system.
- The model must be able to capture **information, behaviour** (operations) and **presentation** (inputs and outputs).



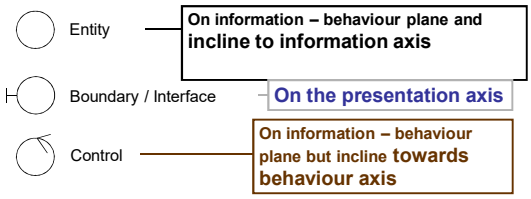
Behaviour - Information - Presentation

- The model is defined in information - behaviour - presentation **space**.

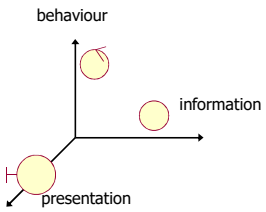


OO Analysis Model Syntax

- Within an use case, we employ three types of **objects**
- In Rational Rose, known as three types of **entities** or **stereotypes**



Entity, Control, Interface



Pragmatics of OO Analysis Model

Identifying interface objects
- functions directly related to **actors**.

Identifying entity objects
- information used in an **use case** and functions of processing the **information**.

Identifying control objects
- functions that **link interface objects** and **entity objects**

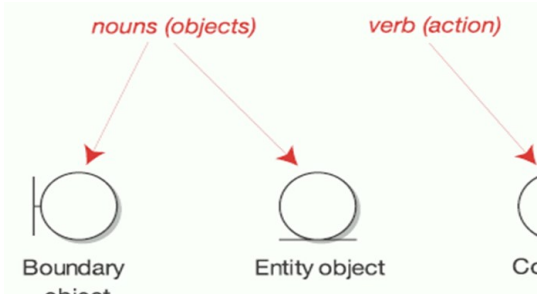
Semantics of OO Analysis Model

- An **entity object** models **information** that shows the state of a system.
  - ✓ This information is often used to **record** the *effects of operations*
  - ✓ Related to the **behaviour** of the **system**.
- A **boundary/interface object** models **inputs** and **outputs** and **operations** that process them.
- A **control object** models **functionality/operations** regarding to validate and decide whether to process and pass information from the interface object to the entity object or the way around.

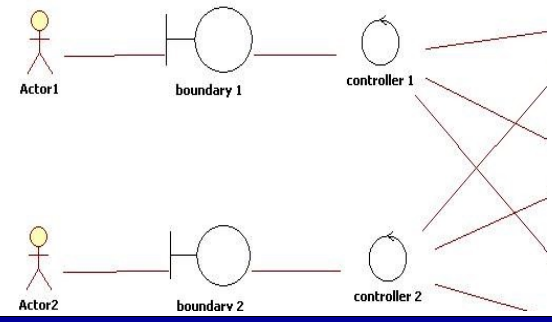
Semantics of OO Analysis Model

- **Interface objects: Mediate** the **communication** with the **actors**. They directly correspond to the actor/system interfaces.
- **Entity objects:** are objects to hold **information**. They often correspond to the **objects** in reality and can be found by ***conceptual domain modeling***.
- **Control objects:** are those objects which **coordinate** and **allocate work** between the different **objects** in fulfillment of a particular **use case.**

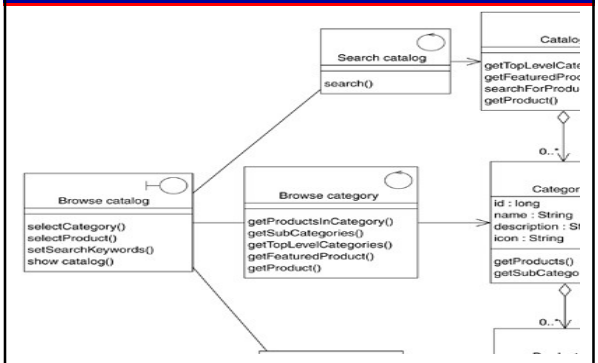
Noun-Verb Conjunction



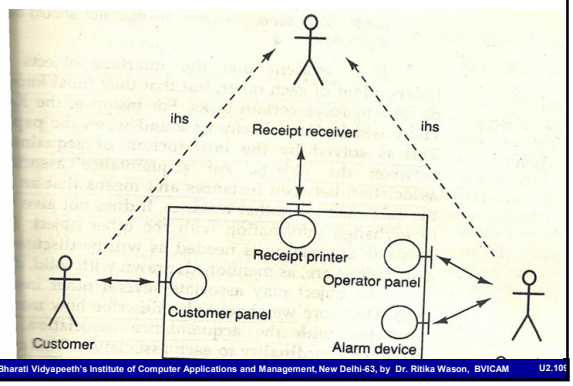
Analysis Model Structure



Another Representation…



Recycling Machine Case Study



Recycling Machine Case Study

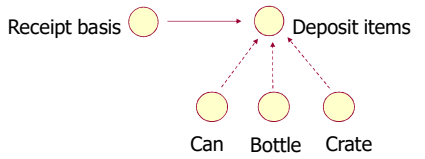
- Identifying interface objects
  - Printer, Customer Panel
- Identifying entity objects
  - Long term information: Crate, Bottle, Can
  - Superclass: Deposit item
  - Short term information: Receipt basis
- Identifying control objects
  - Deposit item receiver

Recycling machine: Interface Objects

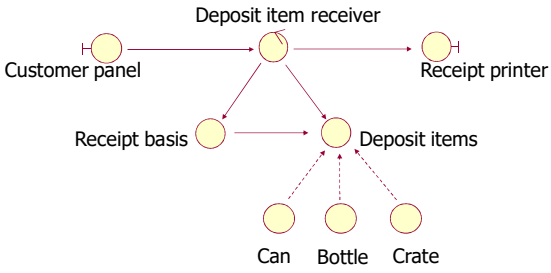
Customer panel

Receipt printer

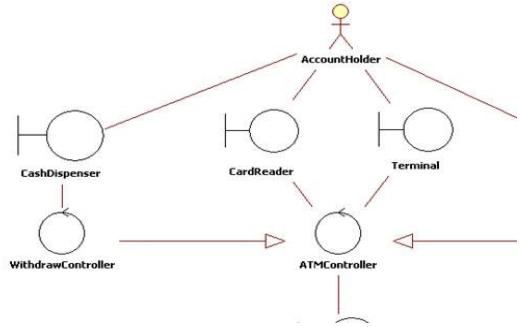
Recycling machine: Entity Objects



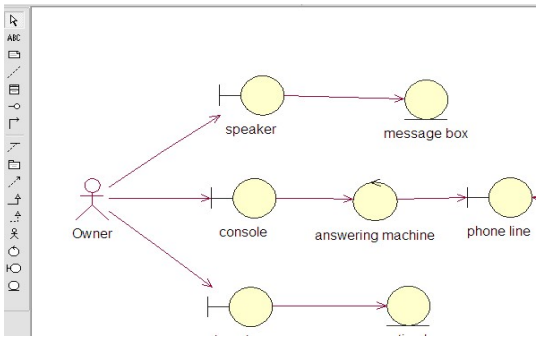
Link Interface and Entity Objects by a Control Object



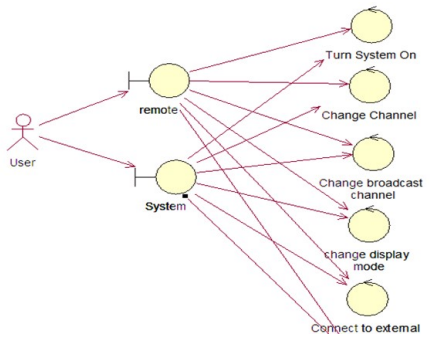
ATM Machine Analysis Model



Telephonic System Analysis Model



Television Control Analysis Model



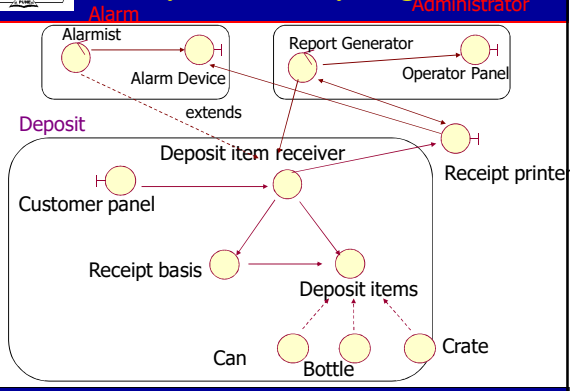
Subsystem in Analysis Model

- **Package** the **objects** so that **complexity** is **reduced**
- Lowest level is **service package**
- **Little communication** between different **subsystem** as possible

Subsystem in Analysis Model

- The aim is to have **strong functional coupling within** the **subsystem** and a **weak coupling between subsytem**
  - Whether two objects are strongly functionally?
  - Will changes in one object lead to changes in the other object?
  - Do they communicate with the same actor?
  - Are both of them dependent on a third object, such as an interface object or an entity object?
  - Does one object perform several operations on the other?

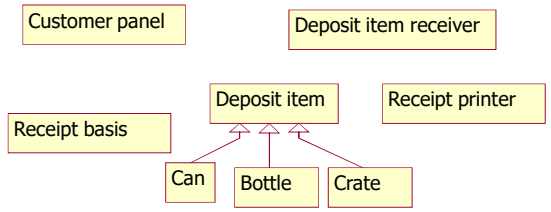
Subsystem in Recycling machine



Good Analysis Class Characteristic

- **Name** reflects its goal
- **Hard abstraction** that models one **specific element** of the problem domain
- **Maps** to a **clearly identifiable feature** of the problem domain
- Has a **small**, **well-defined** set of responsibilities

Drawing of Concepts

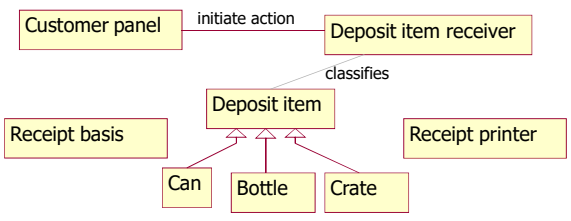


Adding Associations

- If one concept needs to know of a another concept for some duration they should be linked by an association.
- Also use the Common Association list in order to identify associations.

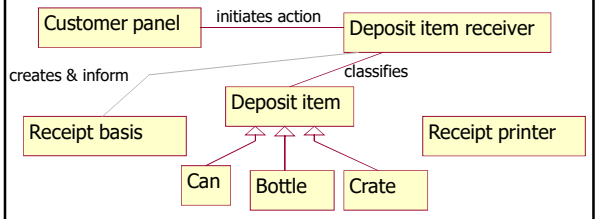
Adding Associations

The Deposit item receiver manages Deposit items:
The items are classified.



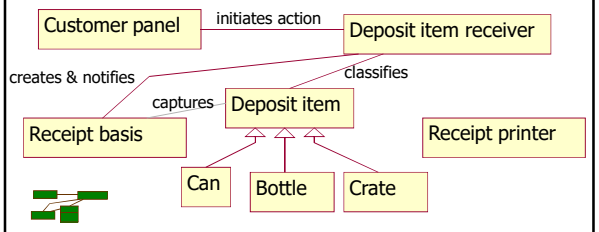
Adding Associations

- The Deposit item receiver communicates to Receipt basis:
- Items received and classified are stored.
- It also creates the receipt basis when it is needed for the first time.



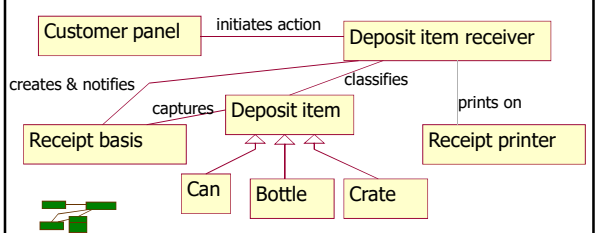
Adding Associations

The Receipt basis collects Deposit items.



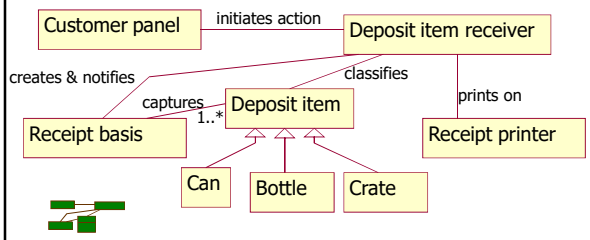
Adding Associations

- On request by the Customer Panel the Deposit item receiver initiates printing of a receipt on the printer.



Adding Associations

- Adding multiplicities
- Only one association here is a 1 to many relationship
- All others are 1 to 1.

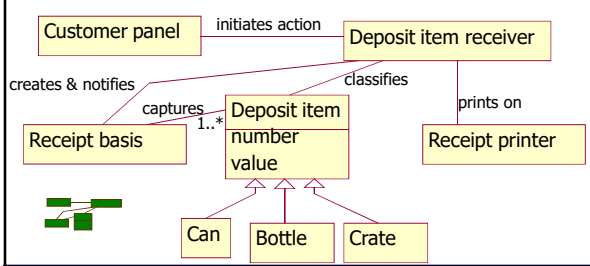


Adding Attributes

- An attribute is a **logical data value** of an **object**.
- Attributes in a conceptual model are **simple data values** as
  - Boolean, Date, Number, String (Text), Time, Address, Colour, Price, Phone Numbers, Product Codes, etc.
- Sometimes it is difficult to distinguish between attributed and concepts
  - E.g. Concept "Car" vs. attribute "Reg. Number".

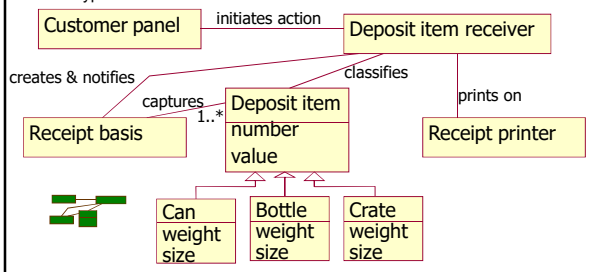
Adding Attributes

- The Deposit item has a value.
- Also it will be assigned a number that shows later on the receipt.



Adding Attributes

- In order to be classified by the Deposit item receiver each item has also a weight and a size.
- However this is the same for each type of item, but different *between* the types.



Conceptual Model

- Representation of concepts in a problem domain.
- **In UML it is basically a "class diagram" without operations.**
- It may show
  - ✓Concepts
  - ✓Associations
  - ✓Attributes

The Concept Category List

- physical or tangible objects
- specifications, designs, descriptions of things
- places
- transactions
- transaction line items
- roles of people
- abstract noun concepts
- organisations
- events
- processes
- rules and policies
- catalogues
- records
- services
- manuals, books

Noun Phrase Identification

- Identify the **noun** and **noun expression** in **textual descriptions** of a **problem domain**
- Consider them as **concepts** and **attributes**.
- Mechanical *noun-to-concept mapping* isn't possible
- Words in usual languages are ambiguous (especially English).

The "return item" Use case-Exercise: Find the Nouns

- The system controls a recycling machine for **returnable bottles, cans and crates.** The machine can be used by several customers at the same time and each customer can return all three types of item on the same occasion. The system has to check, for each item, what type has been returned.
- The **system will register how many items each customer returns** and when the customer asks for a receipt, the system will print out what was deposited , the value of the returned items and the total **return sum that will be paid to the customer.**
- An operator also … (not in "return item" Use Case)

Case Study: Nouns found in the description

- recycling machine
- bottles, cans, and crates
- customers, customer
- types of item, item, type, returned items
- system
- receipt
- return sum

Case Study: Discussion of "recycling machine".

recycling machine
bottles, cans and crates
machine
customers, customer
types of item, item, type, returned items
system
receipt
return sum

- This concept is the "overall system"
- As we consider only one single use case,
- better to name this concept in the context of this use case, e.g.
  - **Deposit item receiver**

Case Study: Discussion of "bottles, cans, and crates".

deposit item receiver
bottles, cans, and crates
machine
customers, customer
types of item, item, type returned items
system
receipt
return sum

- Usually better to use singular and multiplicities instead of plural.
- As **bottle**, **can** and **crate** have much in common (they are processed as items),
- they could be generalised to an "item". We should remember this for later (**inheritance**).

Case Study: Discussion of "machine" and "system"

deposit item receiver
bottle, can, crate
machine
customers, customer
types of item, item, type, returned items
system
receipt
return sum

"Machine" and "System" mean here the same, namely the "Recycling machine", i.e. the
- **Deposit item receiver**

Case Study: Discussion of "customers" and "customer".

deposit item receiver
bottle, can, crate

customers, customer
types of item, item, type, returned items

receipt
return sum

- The customer has already been identified as an actor.
- They are outside of the system.
- We establish a concept, that interfaces with the customer (and is inside the system):
  - **Customer panel**

Case Study: Discussion of "item" (etc.)

deposit item receiver
bottle, can, crate

customer panel
types of item, item, type, returned items

receipt
return sum

- The items that are inserted in the machine.
- Good candidate as superclass for bottle, can, crate.
- Let's call it
  - **Deposit item**

Case Study: Discussion of "receipt"

deposit item receiver
bottle, can, crate

customer panel
deposit item

receipt
return sum

- The concept that "remembers" all items inserted in the machine.
- To distinguish it from the piece of paper returned to the customer, call it
  - **Receipt basis**

Case Study: Discussion of "return sum"

deposit item receiver
bottle, can, crate

customer panel
deposit item

receipt basis
return sum

- The sum that it is returned to the customer is actually computed by adding up all values of the items stored in the receipt basis.
- The sum itself is only a **primitive data value,** and may therefore not be considered as a concept.

Case Study: Discussion of Other Concepts

deposit item receiver
bottle, can, crate

customer panel
deposit item

receipt basis

- These are the concepts identified by nouns. Did we forget something?
- Check the "Concept Category List" !
- The system "**interfaces**" with the physical object "printer", so we add an interface concept
  - **Receipt printer**

Case Study: Summary of Concepts Identified in the Analysis

deposit item receiver
bottle, can, crate

customer panel
deposit item

receipt basis

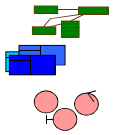
receipt printer

- So far we have identified:
  - Concepts
  - A generalisation relationship.
- **Next step: Finding associations.**

Summary - Object Oriented Analysis

The main task is identifying the objects.
Also: Relationships between objects.
Three strategies:

- Conceptual Model (concepts as objects)
- CRC cards (index cards as objects)
- Analysis Model (Stereotypes as objects)



Next step: Design.

Objective Questions

Q1. Define Architecture.
Q2. Justify the statement System Development is a Model Building.
Q3. At what time, you will decide to start System Development.
Q4. In what way specifications can be used?
Q5. Define Conceptual modeling.
Q6. Define block design.
Q7. Define requirement model.
Q8. Define analysis model.
Q9. Define design model
Q10. Define Implementation Model.
Q11. Define Test Model.

Short Questions

Q1. Suggest some heuristics for identifying objects during object oriented analysis of problem.
Q2. Differentiate between analysis objects with examples.
Q3. Consider air ticket reservation system. Identify entity, control and interface objects.
Q4. Write short note on Architecture.
Q5. Differentiate Method and Process
Q6. What are the five different models for system development, as per the Jacobson approach?
Q7. How models are tightly coupled to the architecture? Discuss.

Long Questions

Q1. What are the features of analysis model and design? Explain with examples.

Q2. For a library management system make analysis model, design model and construction model.

Q3. Justify the statement "System development is model building".

Q4. "The goal if analysis model is to develop a model of what the system will do." Explain the statement with the help of the steps that an analyst will follow throughout the analysis.

Q5. Describe what is done in Analysis with example?

Q6. Describe the system development process with model building.

Research Problems

Q1.Many people invest their money in a number of securities (shares). Generally, an investor has multiple portfolios of investments, each portfolio having investments in many securities. From time to time an investor sells or buys some securities and gets dividends for the securities. There is a current value of each security-many sites give this current value. It is proposed to build a personal investment management system (PIMS) to help investors keep track of their investments as well as on the overall portfolios. The system should also allow an investor to determine the net-worth of the portfolios.

- Discuss the problem analysis for the PIMS problem statement/
- Provide the use case based requirement analysis and specification
- Identify the conceptual objects and draw the Analysis model for PIMS

References

1. Ivar Jacobson, "Object Oriented Software Engineering", Pearson, 2004.
2. Grady Booch, James Runbaugh, Ivar Jacobson, "The UML User Guide", Pearson, 2004
3. R. Fairley, "Software Engineering Concepts", Tata McGraw Hill, 1997.
4. P. Jalote, "An Integrated approach to Software Engineering", Narosa, 1991.
5. Stephen R. Schach, "Classical & Object Oriented Software Engineering", IRWIN, 1996.
6. James Peter, W Pedrycz, "Software Engineering", John Wiley & Sons
7. Sommerville, "Software Engineering", Addison Wesley, 1999.
8. http://www.gentleware.com/fileadmin/media/archives/userguides/poseidon_users_guide/userguide.html
9. http://www.gentleware.com/fileadmin/media/archives/userguides/poseidon_users_guide/statemachinediagram.html
10. http://www.developer.com/design/article.php/2238131/State-Diagram-in-UML.htm

OBJECT-ORIENTED SOFTWARE ENGINEERING

UNIT III

Design and Construction

Learning Objectives

- **Construction**: Introduction, the design model, block design, working with construction. Use case realization: the design discipline within UP iterations.
- **Designing the Subsystem**: Mapping design to code, Designing the data access layer, UI interfaces and system interfaces.
- **Reusable Design Patterns**: Importance of design patterns, Basic design patterns –Singleton, Multiton, Iterator, Adapter, Observer.
- **UML**: Communication Diagrams, Design Class Diagram, State Transition Diagram, Package Diagram, Component Diagram and Deployment Diagram

CONSTRUCTION

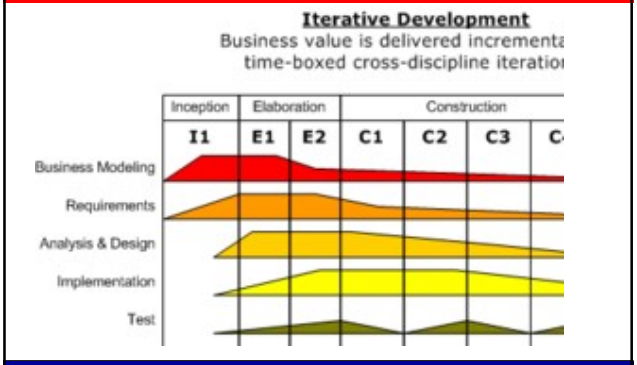
Learning Objectives

- What is Construction Phase
- Why Construction
- Add a Dimension
- Artifacts for Construction
- Design (What, Purpose, Goals, Levels)
- Implementation Environemnt
- Traceability
- Interaction Diagram
- Block design
- Block Behavior
- Implementation

What is Construction Phase?

- All about "**BUILDING**" the **system** from model of analysis & requirement phase.
- Consists of **Design** and **Implementation**.
- Start from **elaboration** & continues to **construction**.

What is Construction Phase?



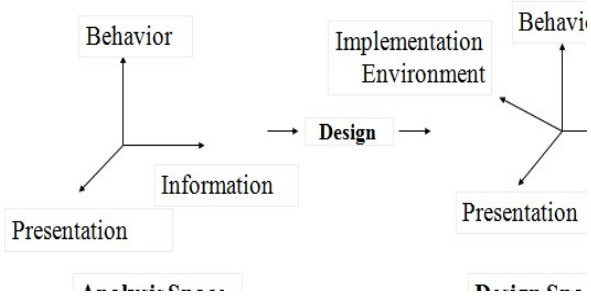
Construction Goals

- The primary goal of the **Construction phase** is to build a system capable of **operating successfully** in beta customer environments.
- During Construction, the **project team** performs tasks that involve building the system **iteratively** and **incrementally** making sure that the viability of the system is always evident in **executable form.**
- The major milestone associated with the Construction phase is called **Initial Operational Capability.**

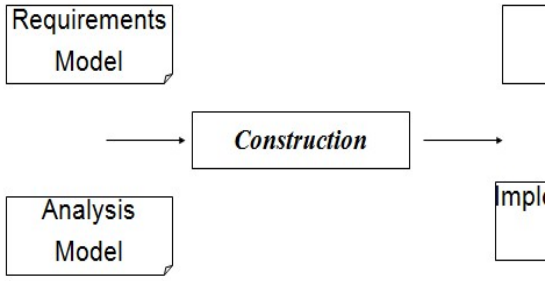
Why Construct?

- For **seamless transition** to **source code**; analysis model is not **sufficient**.
- The actual system must be **adapted** to the **implementation environment.**
- Must explore into more **dimensions.**
- To **validate** the **analysis result**.

Adding A Dimension: Analysis To Design Space



Artifacts of Construction



Design

"*There are two ways of **constructing** a software design*:

- make it so **simple** that there are obviously no deficiencies.
- make it so **complicated** that there are no obvious deficiencies."

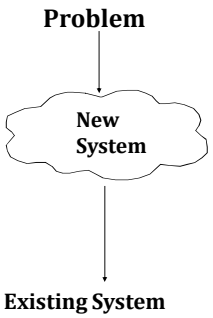
- C.A.R. Hoare

What is Design?

- **Specification** Is about **What**, and **Design** is the start of the **How**?
- **Inputs** to the ***design process***
  - Specification document, including models etc.
- **Outputs** of the ***design process***
  - A design document that describes how the code will be written.
    - What subsystems, modules or components are used
    - How these integrate (i.e. work together)
  - Information allowing **testing** of the system.

Purpose of System Design

- **Bridging** the **gap** between **desired** and **existing** system in a manageable way.
- Use *Divide and Conquer*
- We model the new system to be developed as a **set of subsystems.**



Why is Designing so difficult?

Analysis: Focuses on the application domain

Design: Focuses on the solution domain

- **Design knowledge** is a moving target
- The reasons for design decisions are changing very rapidly
  - ✓Half-time knowledge in software engineering
  - ✓Things will be out of date in 3 years
  - ✓Cost of hardware rapidly sinking

Design Goals

Qualities of a Good Design:

- Correct
- Complete
- Changeable
- Efficient
- Simple

Correctness:

- It Should Lead To A **Correct Implementation**

Completeness:

- It Should Do Everything. Everything? It should follow the **specifications**.

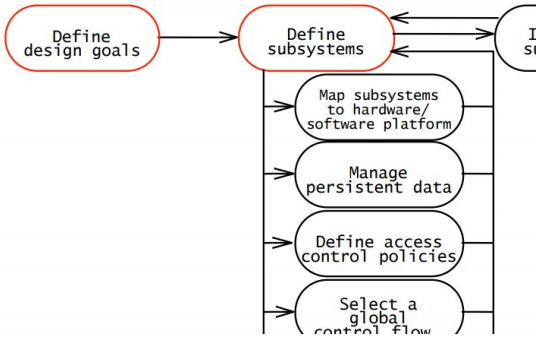
Design Goals

Changeable:
It Should **Facilitate Change**—Change Is Inevitable

Efficiency
- It Should **Not Waste Resources**.
- Better is a **Working Slow Design** Than a Fast Design That Does Not Work.

Simplicity
- It Should Be As **Understandable** As Possible.
- Designs are **blue-prints** for code construction.

Design Goals to Sub-systems



Levels of Design

Three possible levels:
- **System Design**,
  - Part of Systems Engineering.
- **High-level Software Design**
  - Architecture, architectural design.
- **Low-level Software Design**
  - Detailed Design, Module Design.

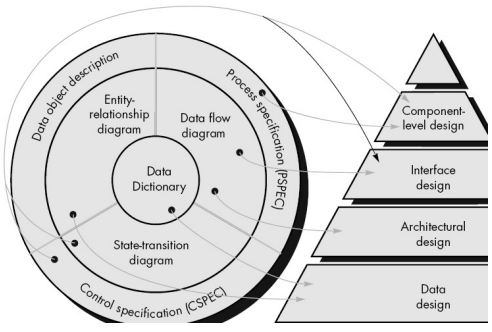
Develop the Design Model

- Create detailed "**plans**" (like **blueprints**) for **implementation.**
- Identify the "*Implementation Environment*" & draw **conclusions**.
- Incorporate the conclusions & develop a "*First approach to a design model*" from requirement models.
  - Use **analysis model** as **base** & translate analysis objects to design objects in design model fit for current implementation
  - Why can't this be **incorporated** in analysis model?

Develop the Design Model

- Describe how the "*Object Interact*" in each specific use case & how **stimuli** between objects is **exchanged**.
- Create **design models** before **coding** so that we can:
  - Compare different possible **design solutions**
  - Evaluate **efficiency**, **ease of modification**, **maintainability**, etc.

Analysis to Design Model

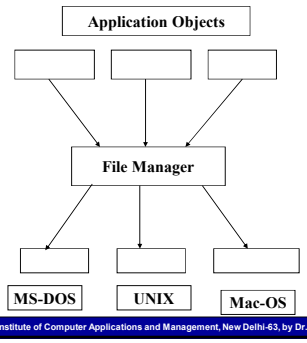


Implementation Environment

- Identify the actual technical constraints under which the system should be built like
  - The **target environment**
  - **Programming language**
  - **Existing products** that should be used (DBMSs, etc)

- **Strategies**:
  - *As few objects as possible* should be aware of the **constraints** of the **actual implementation environment.**

Implementation Env. : Target Env.



Several implementations of the file manager block

Implementation Environment

Target environment

- Create a new blocks that represent occurre changed parts in the target environment

Strategies:

- Specified an abstract class
  - ✓polymorphism
- The object can check the platform at run-time
  - ✓CASE statement in the source code
- Decide this when the system us delivered
  - ✓Provide several different modul which will be ch

- **Investigate whether the target environment will**

Implementation Environment

Programming language

- Affect the design in translating the concepts us
- The basic properties of the language and its are fundamental for the design
  - ✓Inheritance and Multiple inheritance
  - ✓Typing
  - ✓Standard
  - ✓Portability
  - ✓Strategies for handling errors during run-tim
    - **Exception (Ada)**
    - **Assertions (Eiffel)**
  - ✓Memory management

Implementation Environment

Using existing products

- DBMS
- UIMS (User Interface Management System)
- Network facilities
- Internally or externally developed applications tha incorporated
- Products used during development
  - ✓**Compilers**
  - ✓**Debuggers**
  - ✓**Preprocessor**

Other considerations

Implementation Environment

- Other considerations
  - Strategies:
    - ✓To postpone optimizations until they are neede absolutly sure that they will be needed
      - the real bottlenecks are often missed ar optimizations are necessary
  - Use simulation or prototyping to investiga optimization problem early
    - ✓Extensive experiences may help to jugde at an
- If you're not sure of the correctnessof a

Implementation Environment

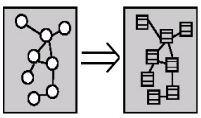
- The people and organization involvec development could also afect the design
  - The principal strategy:
    - ✓such factors should not affect the systen
    - ✓The reason: the circumtances (org staffing, competence areas) that are in c

Traceability

- **Refines** the **analysis model** in light of actual *implementation environment*.
- **Explicit definition** of interfaces of objects, semantics of operation. Additionally, different issues like DBMS, programming language etc. can be considered.
- The model is composed of "**BLOCKS** " which are the **design objects.**
- *One block is implemented as one class.*

Traceability

- The **blocks** abstract the **actual implementation**.
- **Traceability** is extremely important aspect of the system.
  - **Changes** made will be only local to a module.
  - Provides high **functiona localization** (high cohesion).



Analysis Model: *logical, conceptual, frozen*

Design Model: *a practical abstraction*

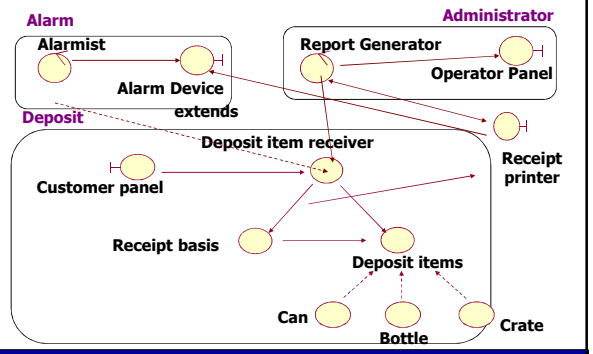
Traceability Matrix

- A **traceability matrix** is a **document**, usually in the form of a **table**, used to assist in **determining** the **completeness of a relationship** by **correlating** any **two baseline documents** using a many-to-many relationship comparison.
- It is often used with **high-level requirements** (these often consist of marketing requirements) and **detailed requirements** of the product to the matching parts of h**igh level design, detailed design, test plan**, and **test cases.**
- A requirements traceability matrix may be used to check to see if the current project requirements are being met, and to help in the creation of a request for proposal, software requirement specification various deliverable documents, and project plan tasks.

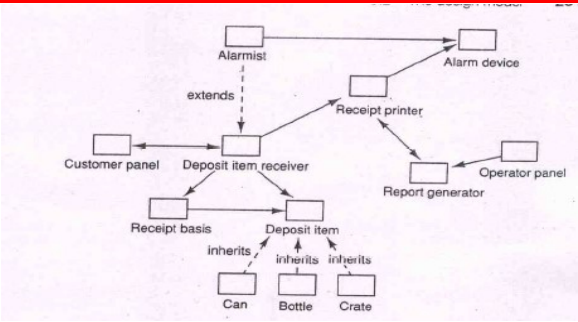
Traceability Matrix

Requirement Identifiers	Reqs Tested	REQ1 UC 1.1	REQ1 UC 1.2	REQ1 UC 1.3	REQ1 UC 2.1	REQ1 UC 2.2	REQ1 UC 2.3.1	REQ1 UC 2.3.2	REQ1 UC 2.3.3	REQ1 UC 2.4	REQ1 UC 3.1	REQ1 UC 3.2	R T
Test Cases	321	3	2	3	1	1	1	1	1	1	2	3	
Tested Implicitly	77												
1.1.1	1	x											
1.1.2	2		x	x									
1.1.3	2	x											
1.1.4	1			x									
1.1.5	2	x											
1.1.6	1		x										
1.1.7	1			x									
1.2.1	2				x		x						
1.2.2	2					x		x					
1.2.3	2								x	x			
1.3.1	1										x		
1.3.2	1										x		
1.3.3	1											x	

Recycling Machine Analysis Model



Recycling Machine Design Model



Working with Design Model

- **Changes** can and should occur, but all changes should be **justified** and **documented** (for *robustness reason*).
- We may have to change the **design model** in various way:
  - To **introduce new blocks** which don't have any representation in the analysis model.
  - To **delete blocks** from the design model.
  - To **change blocks** in the design model (splitting and joining existing blocks).
- To **change** the **associations** between the **blocks** in the design model.

Working with Design Model

- Changes can and should occur, but all changes should be justified and documented (for **robustness** reason).
- We may have to change the design model in various way:
  - To introduce **new blocks** which don't have any representation in the analysis model.
  - To **delete blocks** from the design model.
  - To **change blocks** in the design model (splitting and joining existing blocks).

Change in Environment

- Changing the associations between the blocks in the design model.
  - extensions to stimuli.
  - inheritance to delegation.

Interaction Diagram

- The interaction diagram describes how each **use case** is offered by **communicating objects**
  - ✓The diagram shows how the ***participating objects realize*** the **use case** through their **interaction**
  - ✓The **blocks** send **stimuli** between **one another**
  - ✓All stimuli are **defined** including their **parameters**
- For each **concrete use case**, we draw an **interaction diagram**

Interaction Diagram

- An interaction diagram shows an interaction,
- consisting of a set of objects and their relations
- include the messages that may be exchange them
- Model the dynamic aspect of the system
- Contain two sort of diagrams:
  - **Sequence diagrams**,
    - ✓**show the messages objects send to each timely manner**
  - **Collaboration diagrams**,

Interaction Diagram

- Using interaction diagrams, we can **clarify** the **sequence of operation calls** among **objects** used to complete a single use case
- Collaborations have the added advantage of **interfaces** and **freedom of layout**, but can be difficult to follow, understand and create.
- Interaction diagrams are used to diagram a **single use case**.
- When you want to examine the **behaviour** of a **single instance** over time use a **state diagram**, and if you want to look at the **behaviour** of the system over time use an activity diagram.

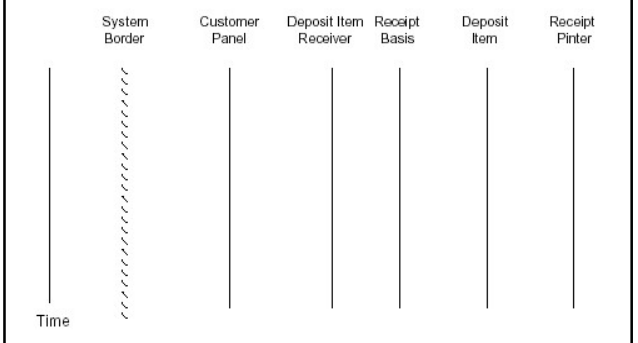
Building an Interaction Diagram

- **Identify blocks**
- Draw **skeleton**, consist of:
  - **System border**
  - **Bars** for each block that participates
- Describes the **sequences**
  - **Structured text** or **pseudo-code**
- Mark the bar to which **operations** belongs with a **rectangle representing operation**
- Define a **stimulus**

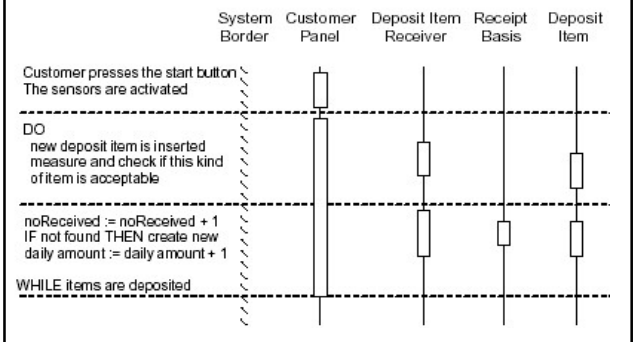
Building an Interaction Diagram

- Draw a **stimulus** as a **horizontal arrow**
  - **Start**: bar of the sending block
  - **End**: bar of the receiving block
- **Structure** the interaction diagram
  - **Fork diagram**
  - **Stair diagram**

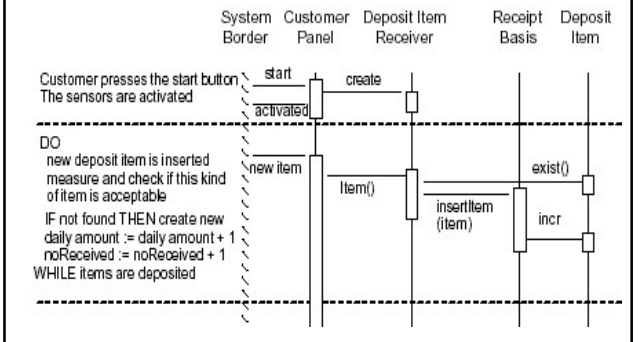
Skeleton of Interaction Diagram



Interaction Diagram for returning item use case



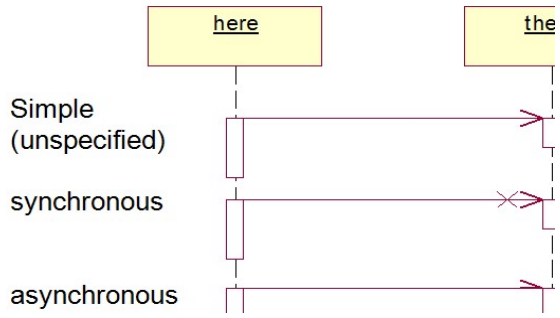
Interaction Diagram for returning item use case



Advanced Interaction Diagram

- A **synchronous message/signal i**s a control which has to *wait for an answer before continuing.*
  - The sender passes the control to the receiver and cannot do anything until the receiver sends the control back.
- An **asynchronous message** is a control which *does not need to wait before continuing.*
  - The sender actually does not pass the control to the receiver.
  - The sender and the receiver carry on their work **concurrently**.

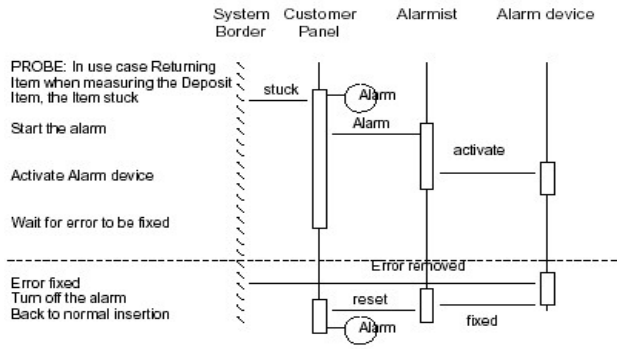
Synchronous and Asynchronous



Probe Condition

- Use case with extension is described by a probe position in the interaction diagram
- The probe position *indicates a position in the use case to be extended*
  - Often accompanied by a **condition** which indicates under what **circumstances** the extension should take place

Example of a Probe Condition



Homogenization

- In **parallel design process**, several **stimuli** with the **same purpose** or meaning are defined by several designers.
- These stimuli should be **consolidated** to obtain as few stimuli as possible.
  - Called *homogenization*.

Example-Homogenization

What_is_your_phone_number?
Where_do_you_live?
Get_address
Get_address_and_phone_number

Homogenized into:

- Get_address
- Get_phone_number

Sequence Diagram

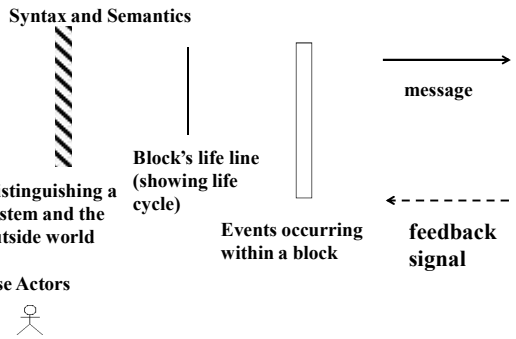
- The sequence diagram describes the flow of messages being passed from object to object.

The **purposes** of **interaction diagram** can be describes as:
- To capture **dynamic behavior** of a system.
- To describe the **message flow** in the system.
- To describe **structural organization** of the objects.
- To describe **interaction** among objects.

Sequence Diagram Elements

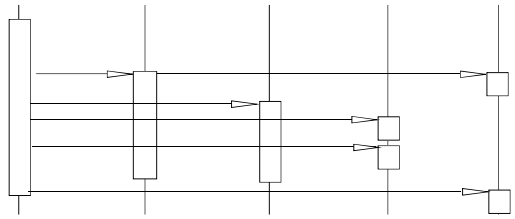
- **Class roles**, which represent **roles** that **objects** may play within the **interaction**.
- **Lifelines**, which represent the **existence** of an **object** over a period of time.
- **Activations**, which represent the **time** during which an object is performing an **operation**.
- The white rectangles on a **lifeline** are called **activations** and indicate that an object is ***responding to a message***. It starts when the message is received and ends when the object is done handling the message.
- **Messages**, which represent **communication** between **objects**.

Sequence Diagram



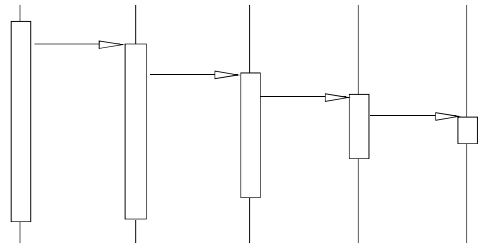
Sequence Diagram- Fork Structure

- **Centralised structure -- Fork: Everything is handled and controlled by the left-most block.**



Sequence Diagram- *Structure*

- **Decentralised structure -- Stair: There is no central control block.**

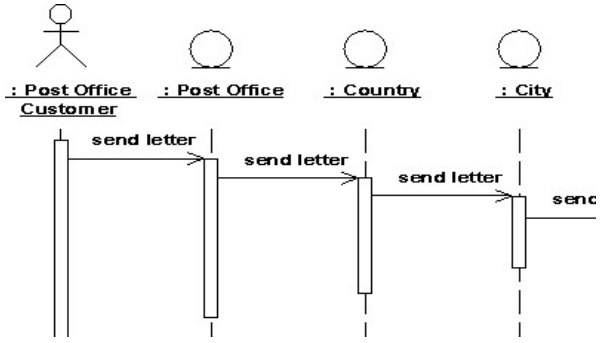


Structure of Sequence Diagram

Decentralized structure is appropriate:

- If the **sub-event phases** are tightly coupled. This will be the case if the participating objects:
  - Form **a part-of** or **consists-of** hierarchy, such as Country - State - City;
  - Form an **information hierarchy**, such as CEO - Division Manager - Section Manager;
  - Represent a **fixed chronological progression** (the sequence of sub-event phases will always be performed in the same order), such as Advertisement - Order - Invoice -Delivery - Payment; or
  - Form a **conceptual inheritance hierarchy**, such as Animal - Mammal - Cat.

Structure of Sequence Diagram

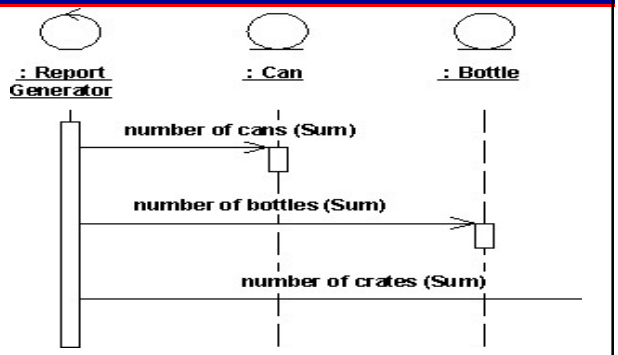


Structure of Sequence Diagram

A centralized structure is appropriate:

- If the order in which the **sub-event phases** will be performed is likely to change.
- If you expect to **insert** new **sub-event phases**.
- If you want to keep parts of the functionality **reusable** as separate piece

Structure of Sequence Diagram



Structure of Sequence Diagram

Fork

- Indicates a **centralized structure** and is characterized by the fact that it is an object controls the other objects interacted with it.
- This structure is appropriate when:
  - The operations can change order
  - New operations could be inserted

Structure of Sequence Diagram

Stair

- Indicates **decentralized structure** and is characterized by **delegated responsibility.**
- Each object only knows a few of the other objects and knows which objects can help with a **specific behavior**.
- This structure is appropriate when:
- The operation have a **strong connection**. Strong connection exists if the objects:
  - form a *'consist-of'* **hierarchy**
  - form an information hierarchy
  - form a fixed temporal relationship
  - form a (conceptual) inheritance relationship
- The operation will always be performed in the same order

Structure Control in Sequence Diagram

- **Optional Execution**
- **Conditional Execution**
- **Parallel Execution**
- **Loop Execution**
- **Nested**

Sequence Diagram

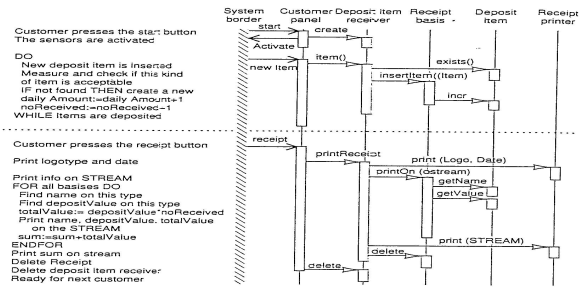


Figure 8.11 Interaction diagram for the use case *Returning Item*.

Block Design

- Block design can start when all the **block** have been identified.
- For block designing it is important to **identify** the **interface and operation** of **each block**.
- The **implementation** (code) for the block can start when the interfaces are **stable** and are **frozen**.
- When the implementation of the block starts, normally **ancestor block** should be implemented prior to **descendent blocks.**

Ex : the deposit item will design prior to can & bottle.

Block Design

- By taking **INTERACTION diagrams** where a block participates & extracting all the operation defined on that block.
- Using this diagram we are clear about the **interface** of the each block..
- The **interface** for **Deposit Item**: exists, incr, getName, getValue

Block Design Comments

- The description of the operation is extracte text to the left of the diagram.
- Can work in parallel once interfaces are fro the open closed principle).

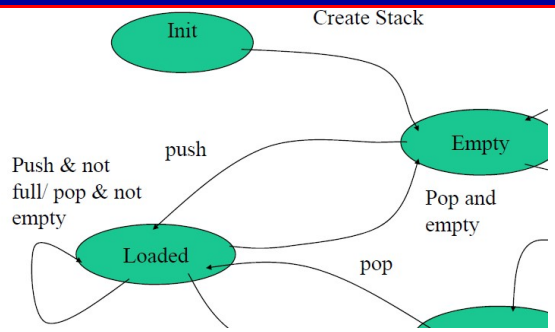
Object Behavior

- An **intermediate level** of **object internal behavior** may be described using a **state machine**.
- To provide a **simplified description** that increases understanding of the block without having to go down to source code.
- State represents **modes of operations** on object.
- **Less dependant** on programming language.
- This is particularly important in **reactive systems**.

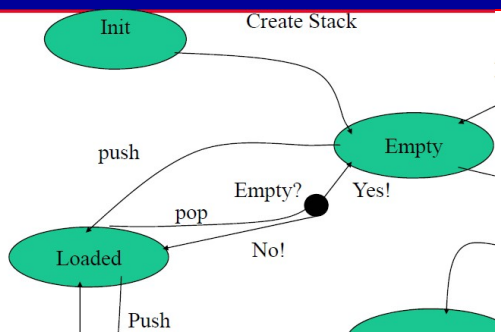
Object Behavior

```
Machine stack
   State init
   input createinstance
   nextstate empty
   otherwise error;
State empty
   input push
   do store on top
   nextstate loaded
   otherwise error;
   ..
endmachine
```

State transition diagram of a stack



State transition diagram of a stack



Stimulus Control Object

Stimulus Control Object

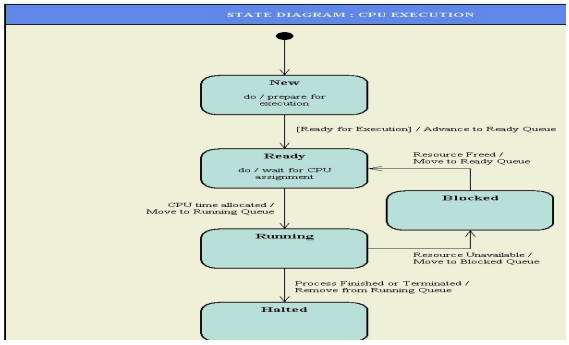
- An object that perform the same operation independent of state when a certain stimulus is received.
- Entity objects

State Controlled Object

State Control Object

- Objects that select operations not only from the stimulus received, but also from the current state
- Control object.

State Controlled Object



Internal Block Structure

- In case of OOPL object-module becomes classes otherwise module unit
- Generally more classes than object
- split class when required
- 5-10 times longer to design a component class than an ordinary class

Implementation

- Now, need to write code for each block.
- Implementation strategy depends on the programming language.
- In an **OOP** language, the implementation of a block starts with one class.
- Sometimes there is a need for additional classes, that are not seen by other blocks.

Mapping

Analysis	Design	Source code C++
Analysis objects	Block	1..N classes
Behavior in objects	Operations	Member functions
Attributes(class)	Attributes(class)	Static variables
Attributes(instance)	Attributes(instance)	Instance variables
Interaction between objects	Stimulus	Call to a function
Use case	Designed use case	Sequence of calls

Implementation Environment

Everything that does not come from analysis phase, including performance requirements.

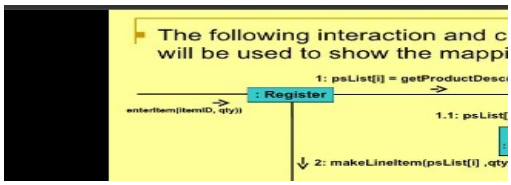
- Design must be adapted to implementation environment.
- Use of existing products must be decided. Includes previous version of the system.
- To use an existing product we must adapt our design.
- Tradeoff - less development vs. more complex architecture.
- Also consider testing costs.

Other Considerations in Construction

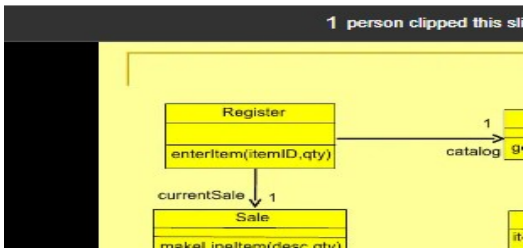
- Subsystems defined in analysis phase are used to guide the construction phase.
- Developed separately as much as possible.
- Incremental development - start construction phase in parallel with analysis phase - to identify implementation environment.
- How much refinement to do in analysis phase? (How early/late to move from analysis to design) - decided in each project

Designing the Subsystem

- The interaction diagrams and the design class diagrams created during design provide some of the necessary input for generating code.
- We now see how to map those artifacts to code in an object-oriented language. The following interaction and class diagram will be used to show the mapping process.



Design Class Diagram



Designing the Data Access Layer

- Design access layer
  - Create mirror classes: For every busine identified and created, create one acces Eg , if there are 3 business classes (clas class2 and class3), create 3 access laye classes (class1DB, class2DB and class3
  - Identify access layer class relationships
  - Simplify classes and their relationships - eliminate redundant classes and structu
    - Redundant classes: Do not keep 2 classes perform similar translate request and transla

Table-Class Mapping

- A tool to map relational data with objects sho following mapping capabilities: (all are two m
- Table-class mapping
- Table-multiple classes mapping
- Table-inherited classes mapping
- Tables-inherited classes mapping
- The tool must describe both how the foreign used to navigate among classes and instanc

Table-Class Mapping

- It is a simple one-to-one mapping of a class and the mapping of columns in a properties in a class. Here we map all to properties. But it is more efficient to those columns for which an object mod required by the application(s). Here ea the table represents an object instance column in the table corresponds to an

Table-Multiple Classes Ma

- Here a single table maps to multiple noninhe classes. Two or more distinct, noninheriting c have properties that are mapped to columns i table. At run time, mapped table row is acces instance of one of the classes, based on a co in the table.
- Table-Inherited Classes Mapping
- Here, a single table maps to many classes th

Tables-Inherited Classes M

- This mapping allows the translation of is-a r that exist among tables in the relational sch class inheritance relationships in the object relational database, an is-a relationship ofte by a primary key that acts as a foreign key t table. In the object-model, is-a is another te inheritance relationship.
- Keys for Instance Navigation
- In mapping columns to properties, the simpl is to translate a column's value into the corr class property value. Here either the colum value or it defines a navigable relationship b

REUSABLE DESIGN PATTERNS

The Beginning of Patterns

- Christopher Alexander, architect
  - A Pattern Language--Towns, Buildings, Construction
  - Timeless Way of Building (1979)
  - "Each pattern describes a *problem* which occurs over and over again in our environment, and then describes the core of the *solution* to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."
- Other patterns: novels (tragic, romantic, crime), movies genres (drama, comedy, documentary)

"Gang of Four" (GoF) Book

- Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Publishing Company, 1994
- Written by this "gang of four"
  - Dr. Erich Gamma, then Software Engineer, Taligent, Inc.
  - Dr. Richard Helm, then Senior Technology Consultant, DMR Group
  - Dr. Ralph Johnson, then and now at University of Illinois, Computer Science Department
  - Dr. John Vlissides, then a researcher at IBM
    - ✓ Thomas J. Watson Research Center
    - ✓ See John's WikiWiki tribute page http://c2.com/cgi/wiki?JohnVliss

Object-Oriented Design Patterns

- This book defined 23 patterns in three categories
  - *Creational patterns* deal with the process of object creation
  - *Structural patterns*, deal primarily with the static composition and structure of classes and objects
  - *Behavioral patterns*, which deal primarily with dynamic interaction among classes and objects

Documenting Discovered Patterns

- Many other patterns have been introduced documented
  - For example, the book **Data Access Patterns** by Clifton Nock introduces 4 decoupling patterns, 5 resource patterns, 5 I/O patterns, 7 cache patterns, and 4 concurrency patterns.
  - Other pattern languages include telecommunications patterns, pedagogical patterns, analysis patterns

GoF Patterns

- *Creational Patterns*
  - ✓ Abstract Factory
  - ✓ Builder
  - ✓ Factory Method
  - ✓ Prototype
  - ✓ Singleton
- *Structural Patterns*
  - ✓ Adapter
  - ✓ Bridge
  - ✓ Composite
  - ✓ Decorator
  - ✓ Façade
  - ✓ Flyweight
  - ✓ Proxy
- *Behavioral Patterns*
  - ✓ Chain of Responsibility
  - ✓ Command
  - ✓ Interpreter
  - ✓ **Iterator**
  - ✓ Mediator
  - ✓ Memento
  - ✓ Observer
  - ✓ State
  - ✓ **Strategy**
  - ✓ Template Method
  - ✓ Visitor

Why Study Patterns?

- Reuse tried, proven solutions
  - Provides a head start
  - Avoids gotchas later (unanticipated things)
  - No need to reinvent the wheel
- Establish common terminology
  - Design patterns provide a common point of reference
  - Easier to say, "We could use Strategy here."
- Provide a higher level prospective
  - Frees us from dealing with the details too early

Other advantages

- Most design patterns make software more modifiable, less brittle
  - we are using time tested solutions
- Using design patterns makes software systems easier to change—more maintainable
- Helps increase the understanding of basic object-oriented design principles
  - encapsulation, inheritance, interfaces, polymorphism

Style for Describing Patterns

- We will use this structure:
  - *Pattern name*
  - *Recurring problem:* what problem the pattern addresses
  - *Solution:* the general approach of the pattern
  - *UML for the pattern*
    - ✓*Participants:* a description as a class diagram
  - *Use Example(s):* examples of this pattern, in Java

A few OO Design Patterns

- **Coming up:**
- **Singleton**
- **Multiton**
  - **Iterator**
    - ✓access the elements of an aggregate object sequentially without exposing its underlying representation
  - **Adaptor**
    - ✓A means to define a family of algorithms, encapsulate each one as an object, and make them interchangeable
  - **Observer**
    - ✓One object stores a list of observers that are updated when the state of the object is changed

Singleton Pattern

6 people clipped this slide

Definition

- Ensures that a class has only one instance, and pr to access this instance.
- In other words, a class must ensure that only singl created and single object can be used by all other

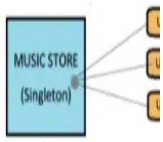
3 people clipped this slide

Advantages and Usage

- Advantages
  - Saves memory because object is not created at each r instance is reused again and again.
  - In cases when object creation is very costly (time takin create new object each time we need it. We just acce

Usage Example

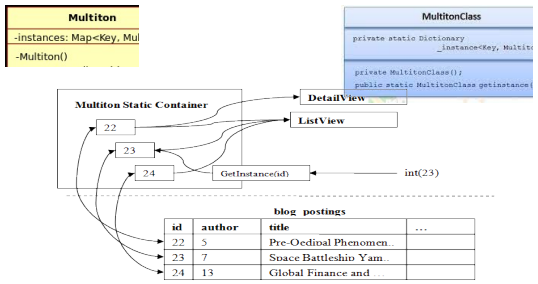
Usage Example



Multiton Pattern

- The **multiton pattern** is a design pattern which generalizes the singleton pattern. Whereas the singleton allows only one instance of a class to be created, the multiton pattern allows for the controlled creation of multiple instances, which it manages through the use of a map.
- Rather than having a single instance *per application* (e.g. the java.lang.Runtime object in the Java programming language) the multiton pattern instead ensures a single instance *per key*.
- *Drawback:* This pattern, like the Singleton pattern, makes unit testing far more difficult, as it introduces global state into an application.
- With garbage collected languages it may become a source of memory leaks as it introduces global strong references to the objects.

Example



Pattern: *Iterator*

- Name: Iterator (a.k.a Enumeration)
- Recurring Problem: How can you loop over all objects in any collection. You don't want to change client code when the collection changes. Want the same methods
- Solution: 1) Have each class implement an interface, and 2) Have an interface that works with all collections
- Consequences: Can change collection class details without changing code to traverse the collection

GoF Version of Iterator page 257

ListIterator
First()
Next()
IsDone()
CurrentItem()

```
// A C++ Implementation
ListIterator<Employee> itr = list.iterator();
for(itr.First(); !itr.IsDone(); itr.Next()) {
   cout << itr.CurrentItem().toString();
```

Java version of Iterator

interface Iterator

boolean hasNext()

Returns true if the iteration has more elements.

Object next()

Returns the next element in the iteration and updates the iteration to refer to the next (or have hasNext() return false)

void remove()

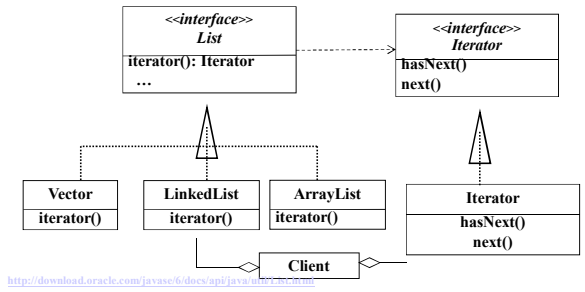
Removes the most recently visited element

Java's Iterator interface

```
// The Client code
List<BankAccount> bank =
                    new ArrayList<BankAccount>();
bank.add(new BankAccount("One", 0.01) );
// ...
bank.add(new BankAccount("Nine thousand", 9000.00));

String ID = "Two";
Iterator<BankAccount> itr = bank.iterator();
while(itr.hasNext())  {
   if(itr.next().getID().equals(searchAcct.getID()))
     System.out.println("Found " + ref.getID());
}
```


UML Diagram of Java's Iterator with a few Collections



http://download.oracle.com/javase/6/docs/api/java/util/Iterator.html

The Observer Design Pattern

- Name: Observer
- Problem: Need to notify a changing number of objects that something has changed
- Solution: Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically

Examples

- From Heads-First: Send a newspaper to all who subscribe
  - People add and drop subscriptions, when a new version comes out, it goes to all currently described
- Spreadsheet
  - Demo: Draw two charts—two views--with some changing numbers--the model

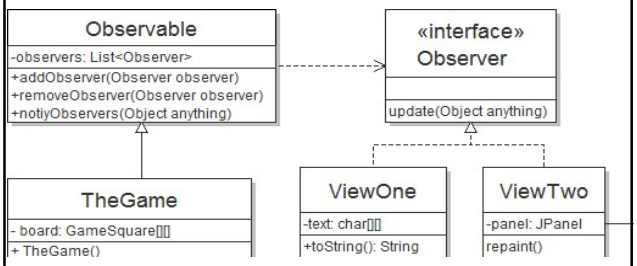
16-107

Examples

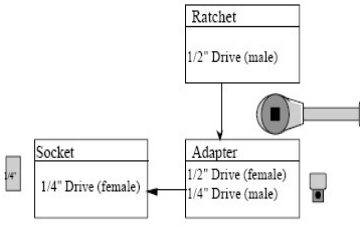
- File Explorer (or Finders) are registered observers (the view) of the file system (the model).
- Demo: Open several finders to view file system and delete a file
- Later in Java: We'll have two views of the same model that get an update message whenever the state of the model has changed

16-108

Observer Example



Adapter (Non software example)



Convert the interface of a class into another Interface clients expect.

Structural

Adapter (Software counterpart)

3 basic building blocks of UML - Diagrams

Graphical representation of a set of elements.

Represented by a connected graph: Vertices are things; Arcs are relationships/behaviors.

5 most common views built from UML 2.0: 12 diagram types

Structural Diagrams

Represent the static aspects of a system.

- Class; Object
- Component
- Deployment

Structural Diagrams

- Class; Object
- Component
- Deployment
- Composite Structure
- Package

Behavioral Diagrams

Represent the dynamic aspects.

- Use case
- Sequence; Collaboration
- Statechart
- Activity

Behavioral Diagrams

- Use case
- Statechart
- Activity

Interaction Diagrams

- Sequence; *Communication*

Class Diagrams

Structural Diagrams

- **Class;** Object
- Component
- Deployment
- Composite Structure
- Package

Class Diagram

The basis for all object modeling
All things lead to this

- **Most common diagram.**
- **Shows a set of classes, interfaces, and collaborations and their relationships (dependency, generalization, association and realization); notes too.**
- **Represents the static view of a system (With active classes, static process view)**

Three modeling perspectives for Class Diagram

- ***Conceptual*: the diagram reflects the domain**
- ***Specification*: focus on interfaces of the software (Java supports interfaces)**
- ***Implementation*: class (logical database schema) definition to be implemented in code and database.**

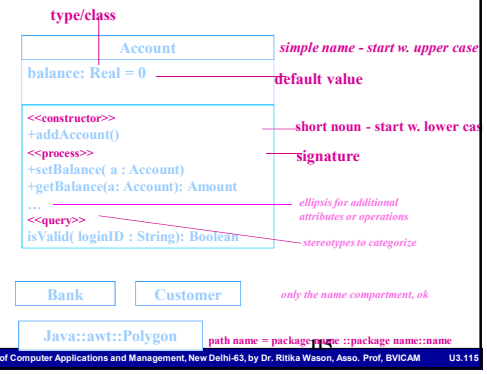
Most users of OO methods take an implementation perspective, which is a shame because the other perspectives are often more useful. -- Martin Fowler

Classes

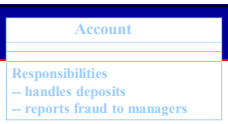
Names

Attributes

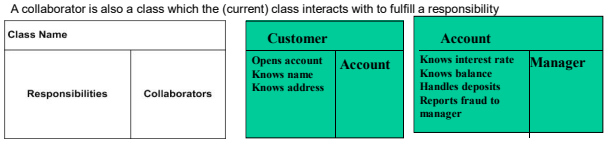
Operations
may cause object to change state



Responsibilities

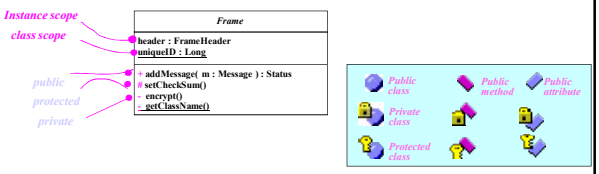


- anything that a class knows or does (Contract or obligation)
- An optional 4th item carried out by attributes and operations.
- Free-form text; one phrase per responsibility.
- Technique - CRC cards (Class-Responsibility-Collaborator); Kent Beck and Ward Cunningham'89
- A collaborator is also a class which the (current) class interacts with to fulfill a responsibility



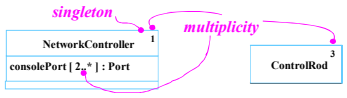
Scope & Visibility

- *Instance* Scope — each instance of the classifier holds its own value.
- *Class* Scope — one value is held for all instances of the classifier (underlined).

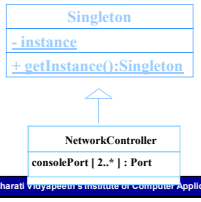


- Public - access allowed for any outside classifier (+).
- Protected - access allowed for any descendant of the classifier (#).
- Private - access restricted to the classifier itself (-).
- (using adornments in JBuilder)

Multiplicity



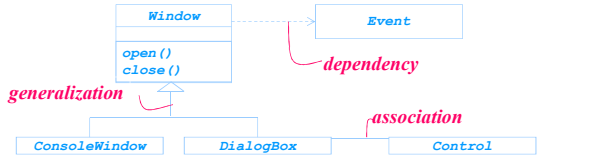
Using Design Pattern

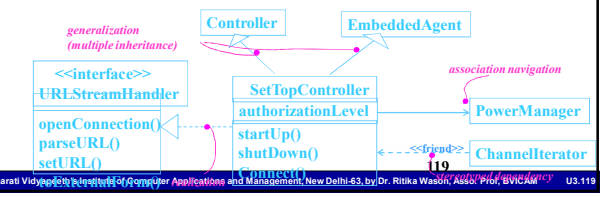



```
public class Singleton {
  private static Singleton instance = null;

  private Singleton() {}
  public static Singleton getInstance() {
    if (instance == null) {
      instance = new Singleton();
    }
    return instance;
  }
}
```


Relationships

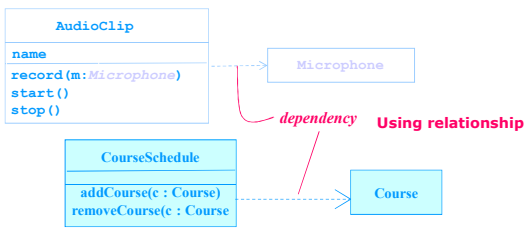




119

Dependency

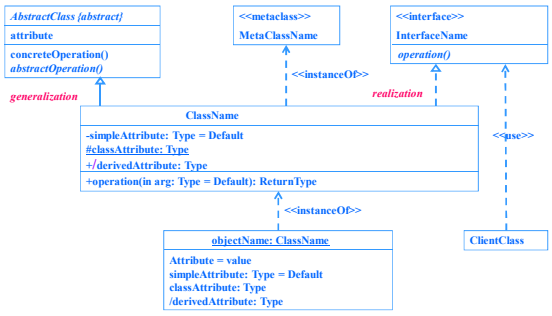
- A change in one thing may affect another.
  - **The most common dependency between two classes is one where one class <<use>>s another as a *parameter to an operation*.**



Usually initial class diagrams will not have any significant number of dependencies in the beginning of analysis but will as more details are identified.

120

Dependency – Among Classes

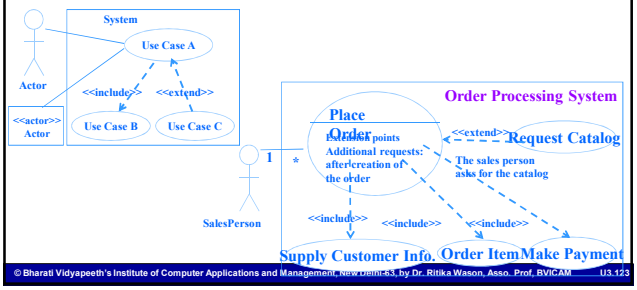


Dependency –Among Classes

- Eight Stereotypes of Dependency Among Classes
  - *bind*: the source instantiates the target template using the given actual parameters
  - *derive*: the source may be computed from the target
  - *friend*: the source is given special visibility into the target
  - ***instanceOf***: the source object is an instance of the target classifier
  - *instantiate*: the source creates instances of the target
  - *powertype*: the target is a powertype of the source; a powertype is a classifier whose objects are all the children of a given parent
  - *refine*: the source is at a finer degree of abstraction than the target
  - *use*: the semantics of the source element depends on the semantics of the public part of the target

Dependency –Among Use Cases

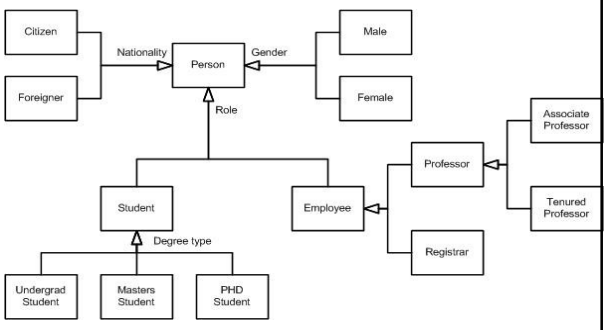
- Two Stereotypes of Dependency Among Use Cases:
  - *extend*: the target use case extends the behavior of the source
  - *include*: the source use case explicitly incorporates the behavior of another use case *at a location* specified by the source



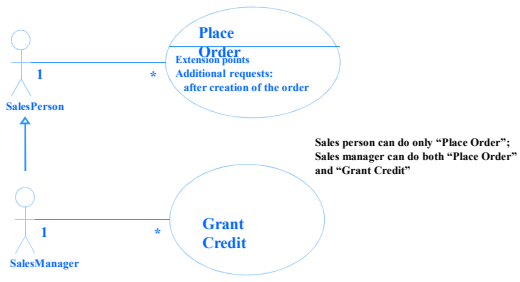
Generalization

- Four Standard Constraints
  - *complete*: all children in the generalization have been specified; no more children are permitted
  - *incomplete*: not all children have been specified; additional children are permitted
  - *disjoint*: objects of the parent have no more than one of the children as a type
  - *overlapping*: objects of the parent may have more than one of the children as a type
- One Stereotype
  - *implementation*: the child inherits the implementation of the parent but does not make public nor support its interfaces

Generalization – Along Roles

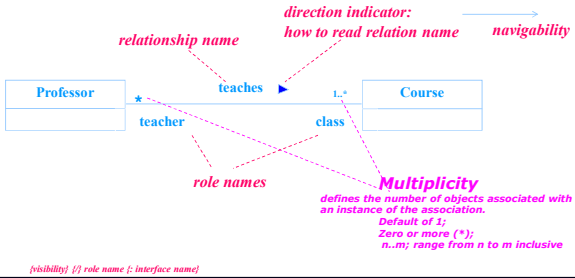


Generalization –Among Actors



Associations

- **Represent conceptual relationships between classes**
  (cf. dependency with no communication/message passing)

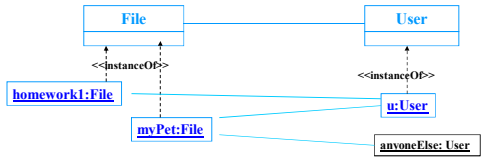


Associations – A Question

- How would you model the following situation?

"You have two files, say homework1 and myPet, where homework1 is read-accessible only by you, but myPet is write-accessible by anybody."

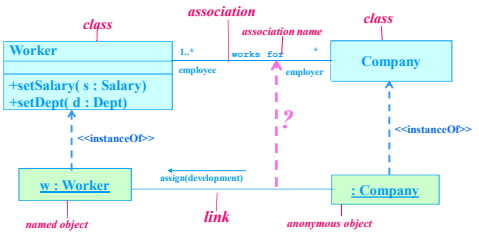
You could create two classes, File and User.
Homework1 and MyPet are files, and you are a user.



Approach 1: Now, would you associate the file access right with File?
Approach 2: Or, would you associate the file access right with User?

Associations – Links

- link is a semantic connection among objects.
- A link is an instance of an association.

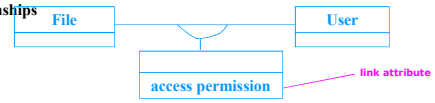


Association generalization is not automatic, but should be explicit in UML

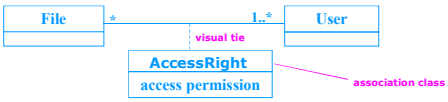
Associations – Link Attributes

- **Link Attributes**

The most compelling reason for having link attributes is for-many-to-many relationships



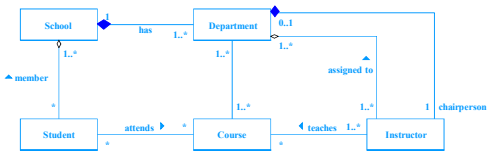
- **Association Class**



- **With a refactoring**

Modeling Structural Relationships

- Considering a bunch of classes and their association relationships



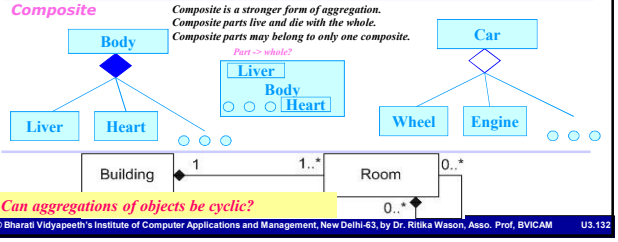
The model above is from Rational Rose. How did the composite symbol ( )get loaded versus the aggregation? **Use the Role Detail and select aggregation and then the "by value" radio button.**

Modeling Structural Relationships

Aggregation - structural association representing "whole/part" relationship.
- "has-a" relationship.

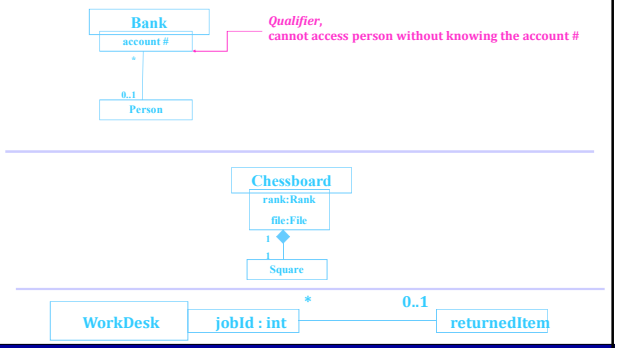
Composite

Composite is a stronger form of aggregation.
Composite parts live and die with the whole.
Composite parts may belong to only one composite.

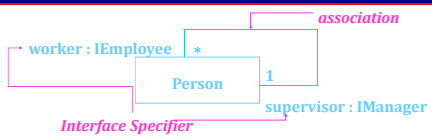


Can aggregations of objects be cyclic?

Association – Qualification



Association – Interface Specifier

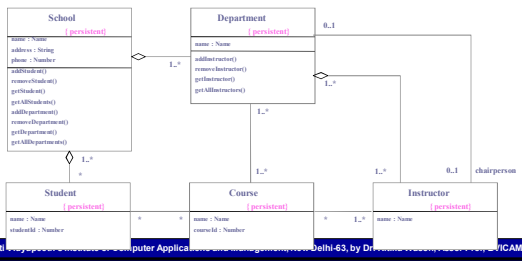


Realization

- **A semantic relationship between classifiers in which one classifier specifies a contract that another guarantees to carry out.**
- **In the context of interfaces and collaborations**
- **An interface can be realized by many classes/components**
- **A class/component may realize many interfaces**

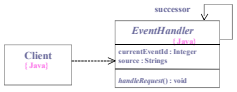
Modeling a Logical Database

- Class diagrams to provide more semantics
- From a general class diagram, first identify classes whose state must be persistent (e.g. after you turn off the computer, the data survives, although the program doesn't).
- Create a class diagram using standard tagged value, (e.g. {persistent}).
- Include attributes and associations.
- Use tools, if available, to transform logical design (e.g., tables and attributes) into physical design (e.g., layout of data on disk and indexing mechanisms for fast access to the data).



Forward/ Reverse Engineering

- translate a collaboration into a logical database schema/operations
- transform a model into code through a mapping to an implementation language.
- Steps
  - Selectively use UML to match language semantics (e.g. mapping multiple inheritance in a collaboration diagram into a programming language with only single inheritance mechanism).
  - Use tagged values to identify language..




```
public abstract class EventHandler
{
    private EventHandler successor;
    private Integer currentEventId;
    private String source;

    EventHandler() {}
    public void handleRequest() {}
}
```

- **translate a logical database schema/operations into a collaboration**
- **transform code into a model through mapping from a specific implementation language.**

Object Diagrams

Structural Diagrams

- Class;
- **Object**
- Component
- Deployment
- Composite Structure
- Package

Instances & Object Diagrams

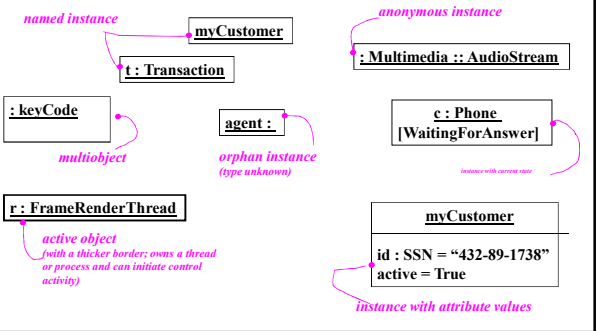
- "instance" and "object" are largely synonymous; used interchangeably.
- difference:
  - instances of a class are called objects or instances; but
  - instances of other abstractions (components, nodes, use cases, and associations) are not called objects but only instances.

What is an instance of an association called?

Object Diagrams

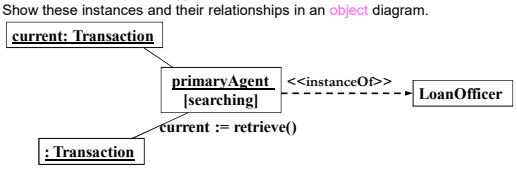
- very useful in debugging process.
  - walk through a scenario (e.g., according to use case flows).
  - Identify the set of objects that collaborate in that scenario (e.g., from use case flows).
  - Expose these object's states, attribute values and links among these objects.

Instances & Objects - Visual Representation



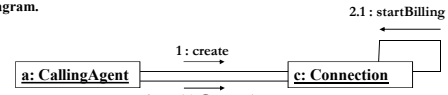
Instances & Objects - Modeling Concrete Instances

- Expose the stereotypes, tagged values, and attributes.
- Show these instances and their relationships in an object diagram.

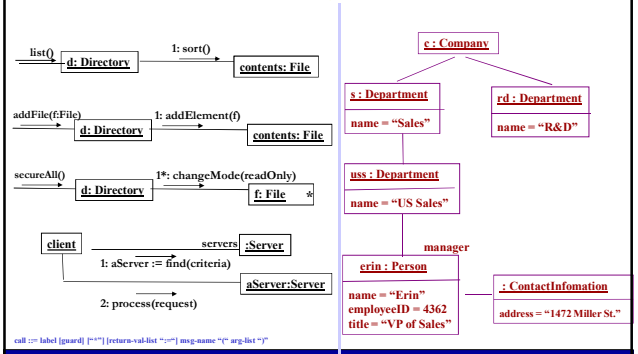


Instances & Objects - Modeling Prototypical Instances

- Show these instances and their relationships in an interaction diagram or an activity diagram.



Instances & Objects - More Examples



Component Diagrams

Structural Diagrams

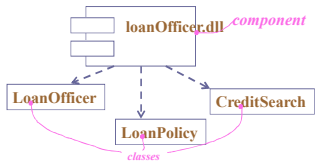
- Class;
  Object
- **Component**
- Deployment
- Composite Structure
- Package

Component Diagram

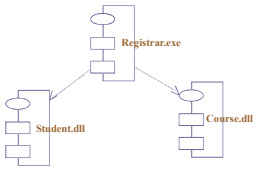
UML1.x – implementation view

- Shows a set of components and their relationships.
- Represents the static implementation view of a system.
- Components map to one or more classes, interfaces, or collaborations.

Mapping of Components into Classes



Components and their Relationships

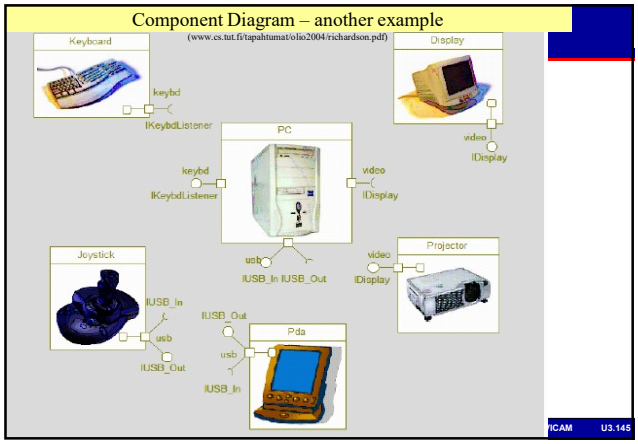


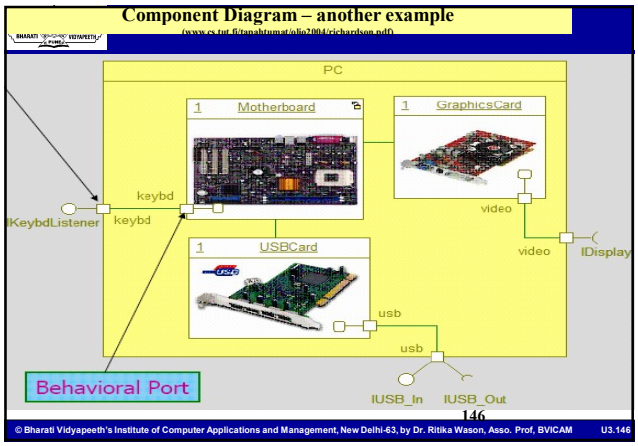
Component Diagram

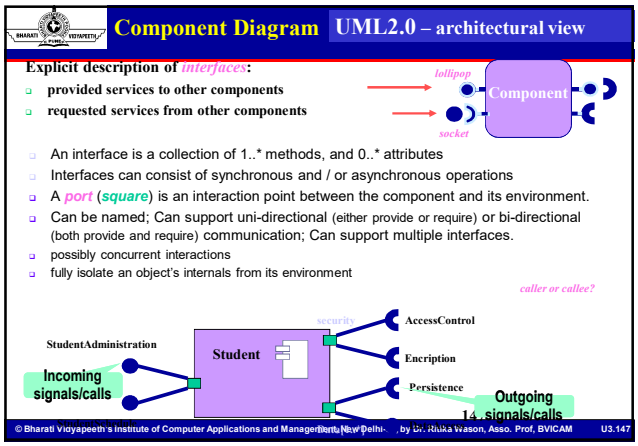
UML2.0 – architectural view

- ***Big demand, hmm…*** ind architecture
- Architecture still an emerging discipline
- Challenges, a bumpy road ahead

- UML and architecture evolving in parallel
- Component diagram in need of better formalization and experimentation

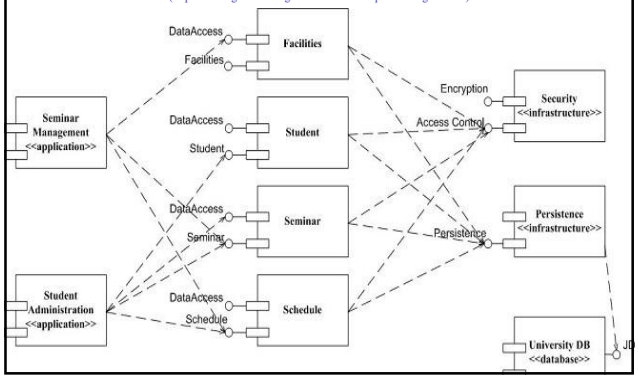






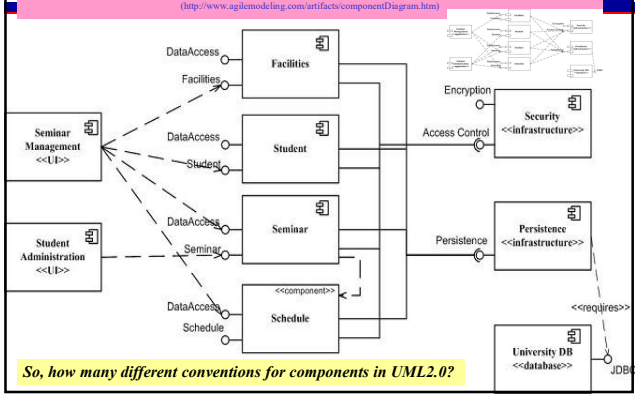
Component Diagram: UML 1.x and UML 2.0

(http://www.agilemodeling.com/artifacts/componentDiagram.htm)



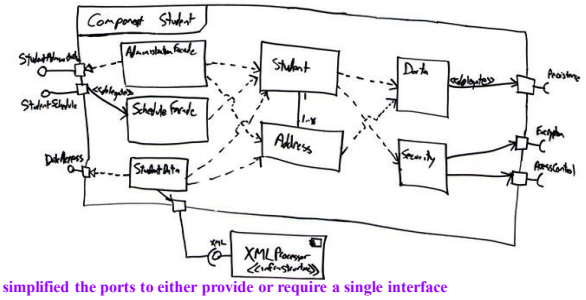
Component Diagram: UML 1.x and UML 2.0

(http://www.agilemodeling.com/artifacts/componentDiagram.htm)



So, how many different conventions for components in UML2.0?

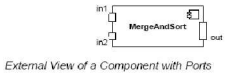
Building a Component



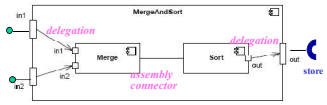
- simplified the ports to either provide or require a single interface
- relationships between ports and internal classes in three different ways:
  i) as *stereotyped delegates* (flow), as *delegates*, and as *realizes* (logical->physical) relationships
- Cohesive reuse and change of classes; acyclic component dependency ???

Component Diagram – Connector & Another Example

- a connector: just a link between two or more connectable elements (e.g., ports or interfaces)
- 2 kinds of connectors: *assembly* and *delegation*. For *"wiring"*
  - An *assembly* connector: a binding between a provided interface and a required interface (or ports) that indicates that one component provides the services required by another; *simple line/ball-and-socket/lollipop-socket notation*
  - A *delegation* connector binds a component's external behavior (as specified at a port) to an internal realization of that behavior by one of its parts *(provide-provide, request-request).*



External View of a Component with Ports



Internal View of a Component with Ports

Left delegation: direction of arrowhead indicates "provides"

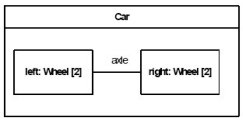
Right delegation: direction of arrowhead indicates "requires"

So, what levels of abstractions for connections?

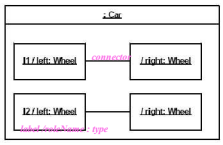
Structured Class

- A *structured class(ifier)* is defined, in whole or in part, in terms of a number of *parts* - contained instances owned or referenced by the structured class(ifier).
- With a similar meaning to a composition relation
- A structured classifier's parts are created within the containing classifier (either when the structured classifier is created or later) and are destroyed when the containing classifier is destroyed.
- Like classes and components, combine the descriptive capabilities of structured classifiers with *ports and interfaces*

Any difference?



component or class?

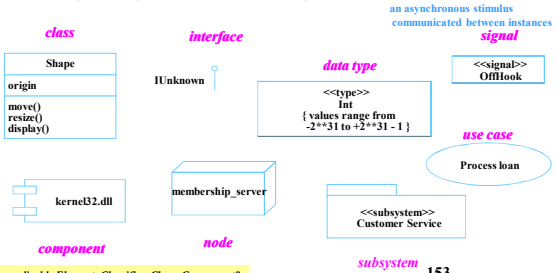


***Components extend classes* with additional features such as**

- **the ability to *own more types of elements* than classes can; e.g., packages, constraints, use cases, and artifacts**
- **deployment specifications that define the execution parameters of a component deployed to a node**

Classifiers

- Classifier—mechanism that describes structural (e.g. class attributes) and behavioral (e.g. class operations) features. In general, those *modeling elements* that can have *instances* are called classifiers.
- *cf. Packages and generalization relationships do not have instances.*



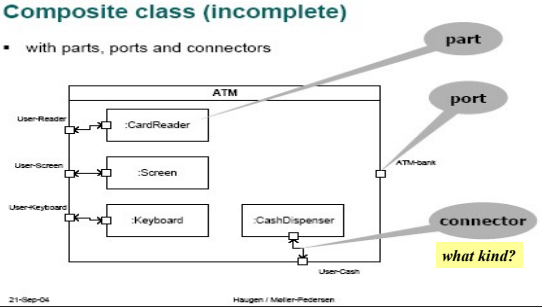
Generalizable Element, Classifier, Class, Component?

Structured Class – Another Example

UNIVERSITY OF OSLO

Composite class (incomplete)

- with parts, ports and connectors



OMG seminar 2004

21-Sep-04 Haugen / Moller-Pedersen 13

Deployment Diagrams

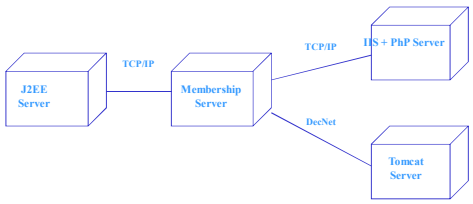
Structural Diagrams

- Class;
  Object
- Component
- **Deployment**
- Composite Structure
- Package

155

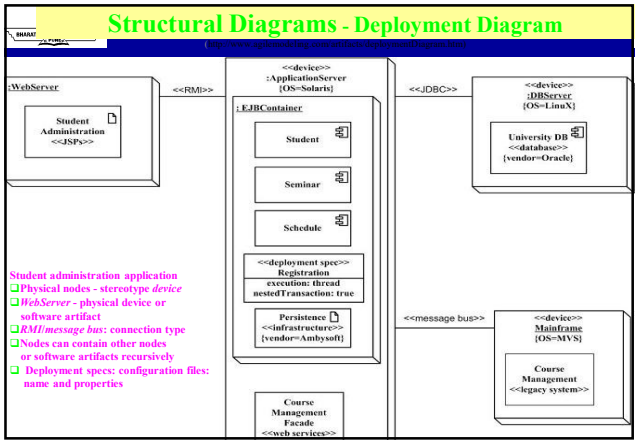
Deployment Diagram

- Shows a set of *processing* nodes and their relationships.
- Represents the static deployment view of an *architecture*.
- Nodes typically enclose one or more components.



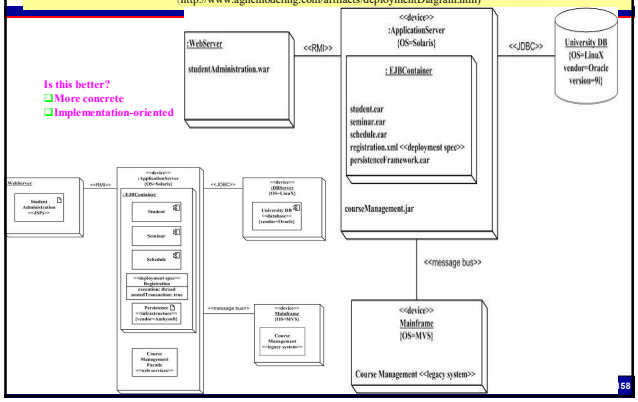
156

Structural Diagrams - Deployment Diagram



Structural Diagrams - Deployment Diagram

(http://www.agilemodeling.com/artifacts/deploymentDiagram.htm)



Composite Structure Diagrams

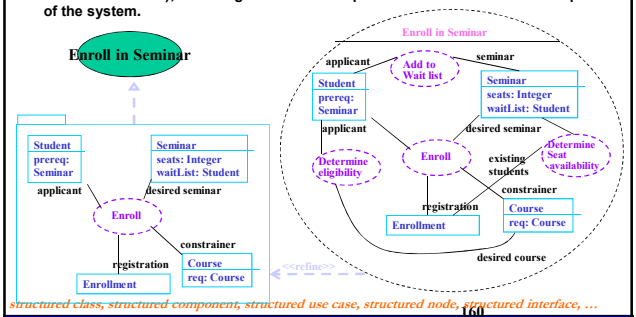
Structural Diagrams

- Class; Object
- Component
- Deployment
- **Composite Structure**
- Package

Composite Structure Diagrams

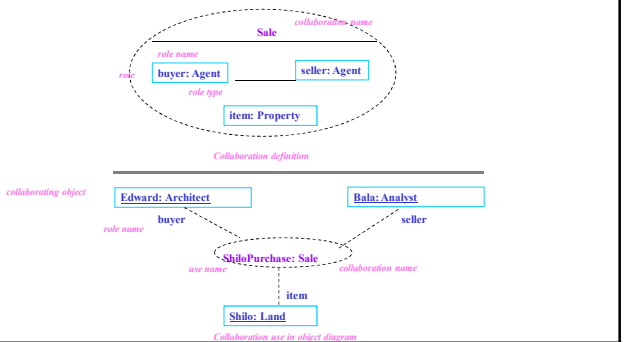
(http://www.agilemodeling.com/artifacts/compositeStructureDiagram.htm)

- Depicts the internal structure of a classifier (such as a class, component, or collaboration), including the interaction points of the classifier to other parts of the system.



structured class, structured component, structured use case, structured node, structured interface, …

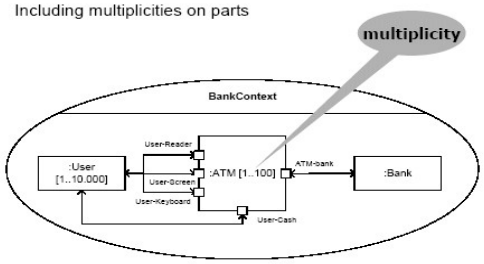
Variations [Rumbaugh – UML 2.0 Reference: p234]



UNIVERSITY OF OSLO

Context Model in UML2.0 - II

- Including multiplicities on parts



OMS seminar 2004

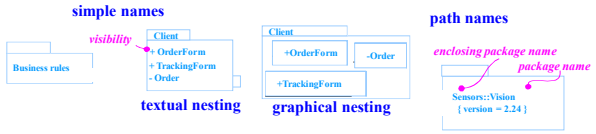
21-Sep-04 Haugen / Møller-Pedersen 15

Structural Diagrams

- Class;
  Object
- Component
- Deployment
- Composite Structure
- Package

Packages

- Package — general-purpose mechanism for organizing elements into groups.
- **Nested Elements: Composite relationship (When the whole dies, its parts die as well, but not necessarily vice versa)**
- **(C++ namespace; specialization means "derived")**



Visibility

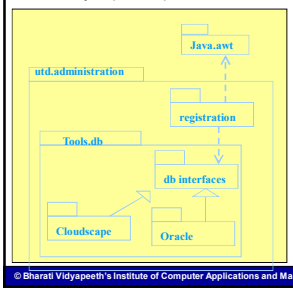
- **Packages that are friends to another may see all the elements of that package, no matter what their visibility.**
- **If an element is visible within a package, it is visible within all packages nested inside the package.**

Dependency –Among Packages

- Two Stereotypes of Dependency Among Packages:
  - *access*: the source package is granted the right to reference the elements of the target package *(:: convention)*
  - *import*: a kind of access; the ***public*** contents of the target package enter the flat namespace of the source as if they had been declared in the source

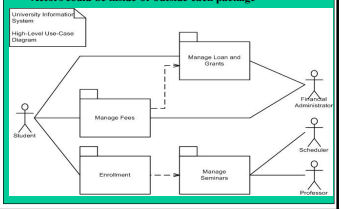
Modeling Groups of Elements

- Look for "clumps" of elements that are semantically close to one another.
- Surround "clumps" with a package.
- Identify public elements of each package.
- Identify import dependencies.



Use Case package Diagram

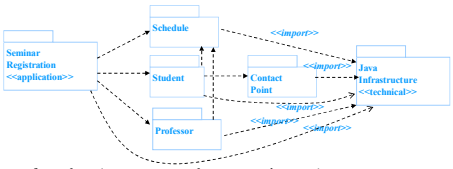
- Included and extending use cases belong in the same package as the parent/base use case
- Cohesive, and goal-oriented packaging
- Actors could be inside or outside each package



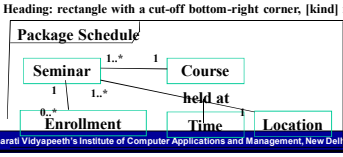
Class Package Diagrams

(http://www.agilemodeling.com/artifacts/packageDiagram.htm)

- Classes related through inheritance, composition or communication often belong in the same package



- A *frame* depicts the contents of a package (or components, classes, operations, etc.)
- Heading: rectangle with a cut-off bottom-right corner, [kind] name [parameter]



A frame encapsulates a collection of collaborating instances or refers to another representation of such

Common Mechanisms

- **Adornments**
  - **Notes & Compartments**
- **Extensibility Mechanisms**
  - **Stereotypes** - Extension of the UML metaclasses.
  - **Tagged Values** - Extension of the properties of a UML element.
  - **Constraints** - Extension of the semantics of a UML element.

168

Adornments

- Textual or graphical items added to an element's basic notation.
- Notes - Graphical symbol for rendering constraints or comments attached to an element or collection of elements; No Semantic Impact

Rendered as a rectangle with a dog-eared corner.

See smartCard.doc for details about this routine.

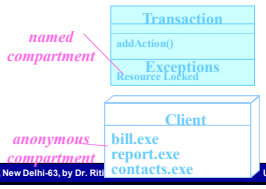
May contain combination of text and graphics.

See http://www.rational.com for related info.

May contain URLs linking to external documents.

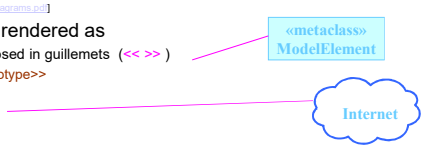
Additional Adornments

- Placed near the element as
  - Text
  - Graphic
- Special compartments for adornments in
  - Classes
  - Components
  - Nodes



Stereotypes

- **Mechanisms for extending the UML vocabulary.**
- **Allows for new modeling building blocks or parts.**
- Allow controlled extension of metamodel classes.
  [UML11_Metamodel_Diagrams.pdf]
- Graphically rendered as
  - Name enclosed in guillemets (<< >>)
    - ✓ <<stereotype>>
  - New icon



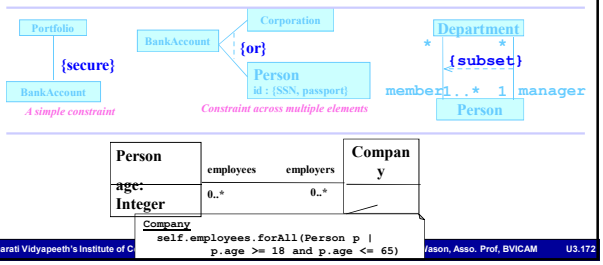
- **The new building block can have**
  - **its own special properties through a set of tagged values**
  - **its own semantics through constraints**

Tagged Values

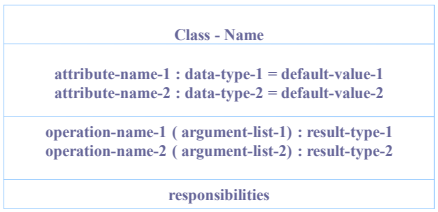
- **a (name, value) pair describes a property of a model element.**
- **Properties allow the extension of "*metamodel*" element attributes.**
- **modifies the semantics of the element to which it relates.**
- **Rendered as a text string enclosed in braces { }**
- **Placed below the name of another element.**

Constraints

- Extension of the semantics of a UML element.
- Allows new or modified rules
- Rendered in braces {}.
  - Informally as free-form text, or
  - Formally in UML's Object Constraint Language (OCL):
    E.g., {self.wife.gender = female and self.husband.gender = male}



Classes: Notation and Semantics



To model the <<semantics>> (meaning) of a class:

- Specify the body of each method (pre-/post-conditions and invariants)
- Specify the state machine for the class
- Specify the collaboration for the class
- Specify the responsibilities (contract)

Attributes

- Syntax
  [visibility] name [multiplicity] [: type] [= initial-value] [{property-string }]
- Visibility
  + public; - private; # protected; {default = +}
- type
  - There are several defined in Rational Rose.
  - You can define your own.

  Or you can define your own: e.g. {leaf}
- property-string
  Built-in property-strings:
  - *changeable*—no restrictions (default)
  - *addOnly*—values may not be removed or altered, but may be added
  - *frozen*—may not be changed after initialization

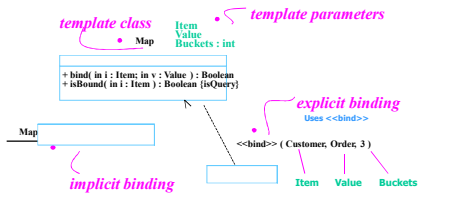
origin	Name only
+ origin	Visibility and name
origin : Point	Name and type
head : *Item	Name and complex type
name [0..1] : String	Name, multiplicity, and type
origin : Point = { 0, 0 }	Name, type, and initial value
id : Integer { frozen }	Name and property

Operations

- Syntax
  [visibility] name [(parameter-list)] [: return-type] [(property-string)]
- Visibility
  + public; - private; # protected; {default = +}
- parameter-list syntax
  [direction] name : type [= default-value]
- direction
  - *in*—input parameter; may not be modified
  - *out*—output parameter; may be modified
  - *inout*—input parameter; may be modified
- **property-string**
  - ***leaf***
  - ***isQuery*—state is not affected**
  - ***sequential*—not thread safe**
  - ***guarded*—thread safe (Java synchronized)**
  - ***concurrent*—typically atomic; safe for multiple flows of control**

Template Classes; Primitive Types

- A template class is a parameterized element and defines a family of classes
- In order to use a template class, it has to be instantiated
- Instantiation involves binding formal template parameters to actual ones, resulting in a concrete class



Primitive Types using a class notation

Interface: A Java Example

```
public interface SoundFromSpaceListener extends EventListener {
    void handleSoundFromSpace(SoundFromSpaceEventObject sfseo);
}

public class SpaceObservatory implements SoundFromSpaceListener
    public void handleSoundFromSpace(SoundFromSpaceEventObject sfseo) {
            soundDetected = true;
            callForPressConference();
    }
```

Can you draw a UML diagram corresponding to this?

Package Diagrams: Standard Elements

- *Façade* — only a view on some other package.
- *Framework* — package consisting mainly of patterns.
- *Stub* — a package that serves as a proxy for the public contents of another package.
- *Subsystem* — a package representing an independent part of the system being modeled.
- *System* — a package representing the entire system being modeled.

Is <<import>> transitive?
Is visibility transitive?
Does <<friend>> apply to all types of visibility: +, -, #?

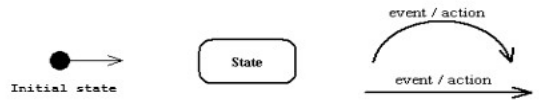
Dependency –Among Objects

- 3 Stereotypes of Dependency in Interactions among Objects:
  - *become*: the target is the same object as the source but at a later point in time and with possibly different values, state, or roles
  - *call*: the source operation invokes the target operation
  - *copy*: the target object is an exact, but independent, copy of the source

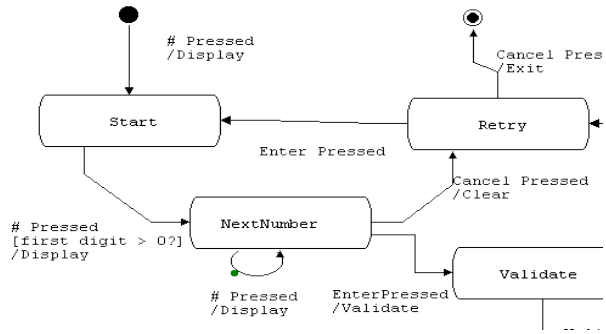
State Machine Diagram

- UML state machine diagrams depict the **various states** that an **object** displays.
- And the **transitions** between those states
- State diagrams show the ***change of an object over time***
- Very useful for **concurrent** and **real-time systems**

State Machine Diagram Notations



State Machine Diagram (*Login*)



State Machine

- "The state machine view describes the **dynamic behavior** of objects over time by **modeling** the **lifecycles** of **objects** of each **class**.
- Each **object** is treated as an **isolated entity** that communicates with the rest of the world by **detecting events** and responding to them.
- Events represent the **kinds of changes** that objects can detect... Anything that can affect an object can be characterized as an event."

  - *The UML Reference Manual, [Rumbaugh,99]*

State Chart Diagram

- It shows a **machine** consisting of **states**, **transitions**, **events** and **activities**.
- It addresses the **dynamic view** of the **system**.
- It is **depicted** as follows in rational rose.

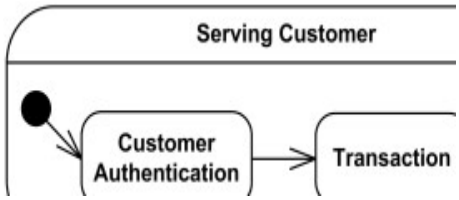
State Diagram Features

- **Event –** something that happens at a **specific point**
  - Alarm goes off
- **Condition –** something that has a **duration**
  - Alarm is on
  - Fuel level is low
- **State** – an abstraction of the **attributes** and relationships of an **object** (or system)
  - The fuel tank is in a **too low level** when the fuel level is **below level *x*** for *n* seconds

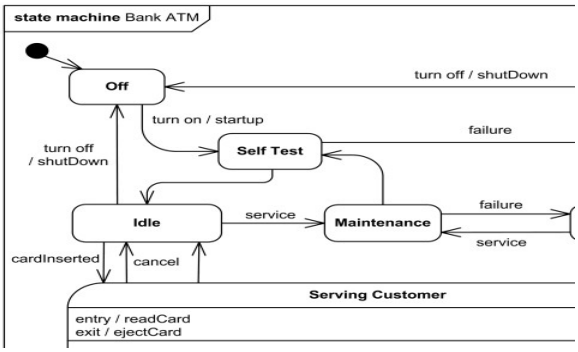
State Diagram Features

- ***State***. A **condition during the life** of an object in which it satisfies some condition, performs some action, or waits for some event.
- ***Event***. An occurrence that may **trigger a state transition.** Event types include an explicit **signal** from outside the system, an invocation from inside the system, the passage of a designated period of time, or a designated **condition becoming true.**
- ***Guard***. A boolean expression which, if true, **enables an event to cause a transition.**
- ***Transition***. A transition represents the change from one state to another:
- ***Action***. One or more actions taken by an object in response to a state change.

Composite States



State Chart Diagram



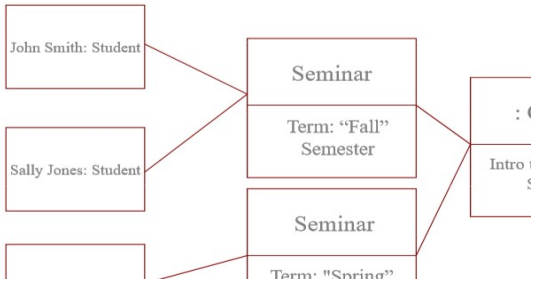
State Diagram- *Case Study*

- A simple digital watch has a display and two buttons A button and the B button. The watch has tw operation, display time and set time.
- In the display time mode, the watch displays hours separated by a flashing colon.
- The set time mode has two submodes, set hours and The A button selects modes. Each time it is presse advances in the sequence: display, set hours, display, etc.
- Within the submodes, the B button advances t minutes once each time it is pressed. Buttons must

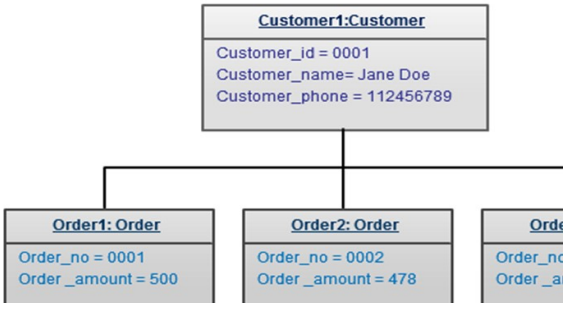
Object Diagram

- Object diagram is a **pictorial representation** of the **relationships** between the **instantiated classes** at any point in time.
- It's depicted as follows in Rational Rose-

Object Diagram Example



Object Diagram Example



Activity Diagram

- Activity diagrams are typically used for **business process modeling**
- For modeling the **detailed logic** of a **business rule**
- Model the **internal logic** of a **complex operation**
- An activity diagram is a special case of a **state chart diagram** in which states are **activities** ("*functions*")
- Activity diagrams are the **object-oriented equivalent** of **flow charts** and data flow diagrams (**DFDs**)

Activity Diagram Structuring

These diagrams are similar to **state chart diagrams** and use similar conventions, but activity diagrams describe the *behavior of a class* in response to **internal processing.**

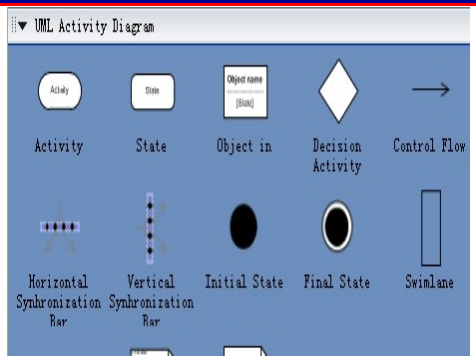
Swimlanes, which represent **responsibilities** of one or more objects for actions within an overall activity; that is, they divide the activity states into groups and assign these groups to objects that must perform the activities.

- **Action States**, which represent **atomic**, or **non-interruptible**, **actions** of entities or steps in the execution of an algorithm.
- **Action flows**, which represent **relationships** between the different action states of an entity.
- **Object flows**, which represent the **utilization of objects** by action states and the influence of action states on objects.

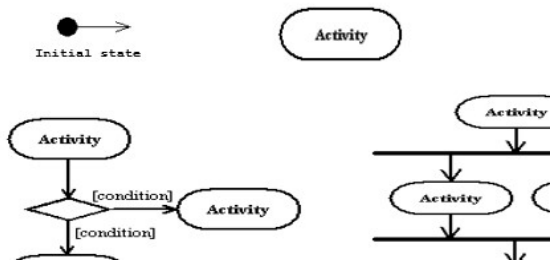
Activity Diagram Notations

- **Rounded rectangles** represent **activities**;
- **Diamonds** represent **decisions**;
- **Bars** represent the start (split) or end (join) **of concurrent activities**;
- **A black circle** represents the start (**initial state**) of the workflow;
- An **encircled black circle** represents the end (**final state**).

Activity Diagram Notations

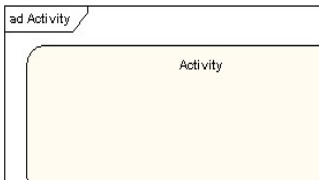


Activity Diagram Notations



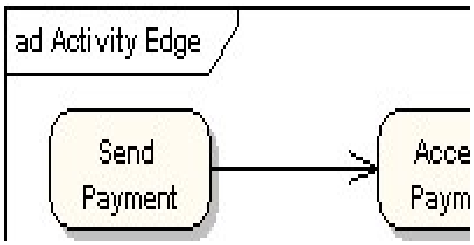
Activities

An activity is the specification of a **parameterized sequence** of **behaviour**. An activity is shown as a **round-cornered rectangle** enclosing all the actions, control flows and other elements that make up the activity.



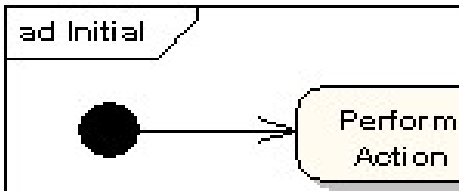
Control Flow

A control flow shows the **flow of control** from one action to the next. Its notation is a **line** with an **arrowhead**.



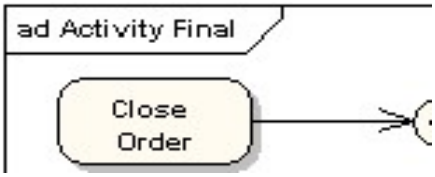
Initial Node

An initial or start node is depicted by a **large black spot**, as shown below.



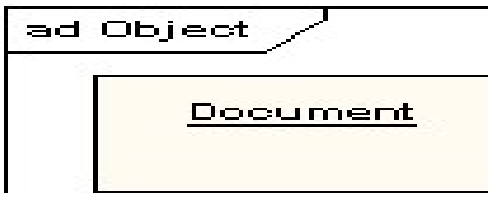
Final Node

The activity final node is depicted as a **circle** with a **dot** inside.



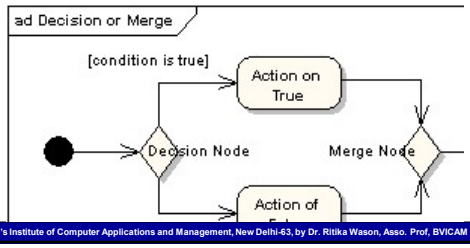
Object and Object Flows

An object flow is a **path** along which objects or data can pass. An object is shown as a **rectangle**.



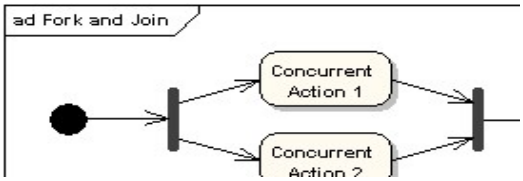
Decision and Merge Nodes

Decision nodes and merge nodes have the same notation: a **diamond shape**. They can both be named. The control flows coming away from a decision node will have **guard conditions** which will allow control to flow if the guard condition is met.



Fork and Join Nodes

Forks and joins have the **same notation**: either a horizontal or vertical bar (the orientation is dependent on whether the control flow is running left to right or top to bottom). They indicate the start and end of **concurrent threads of control**.



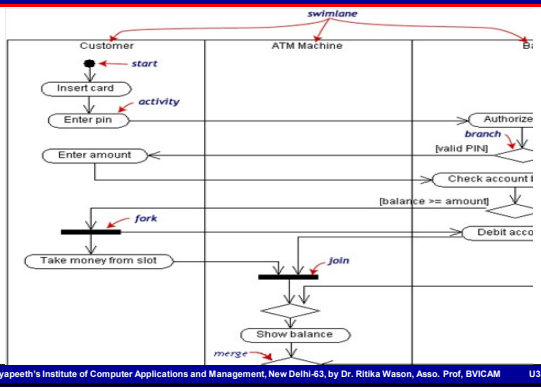
Steps to create an AD

1. Identify activities (steps) of a process
2. Identify who/what performs activities steps)
3. Draw swimlines
4. Identify decision points (if-then)
5. Determine if step is in loop (*For each.* based loop)
6. Determine if step is parallel

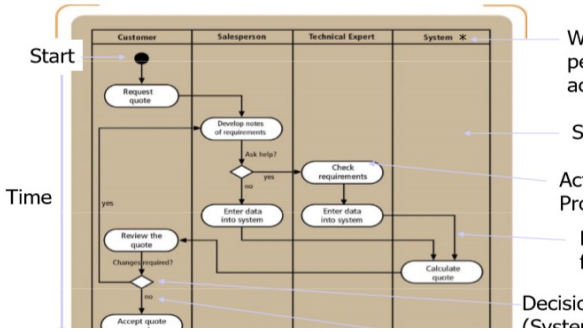
Steps to create an AD

8. Draw the start point of the process in the swir first activity (step)
9. Draw the oval of the first activity (step)
10. Draw an arrow to the location of the second a
11. Draw subsequent activities, while insertin points and synchronization/loop bars where a

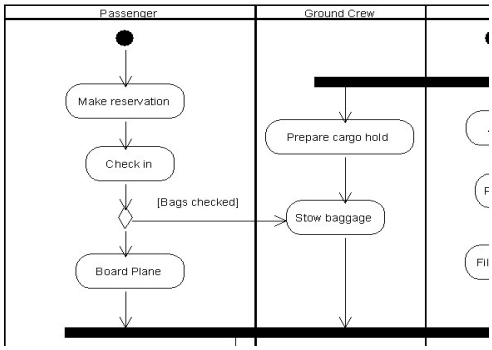
Activity Diagram- ATM Machine



Activity Diagram- Quote System



Activity Diagram- Flight Mgt. System



Dynamic Diagram Types

- Interaction Diagrams - Set of objects or role messages that can be passed among them.
  - Sequence Diagrams - emphasize time order
  - Communication Diagrams - emphasize struc ordering
- State Diagrams
  - State machine consisting of states, transitio and activities of an object
- Activity & Swimlane Diagrams

Interaction Diagram

- An interaction diagram shows an interaction,
- consisting of a set of objects and their relations
- include the messages that may be exchange them
- Model the dynamic aspect of the system
- Contain two sort of diagrams:
  - **Sequence diagrams**,
    - ✓**show the messages objects send to eacl timely manner**
  - **Collaboration diagrams**,

Interaction Diagram Details

- Using interaction diagrams, we can **clarify** the **sequence of operation calls** among **objects** used to complete a single use case
- Collaborations have the added advantage of **interfaces** and **freedom of layout**, but can be difficult to follow, understand and create.
- Interaction diagrams are used to diagram a **single use case**.
- When you want to examine the **behaviour** of a **single instance** over time use a **state diagram**, and if you want to look at the **behaviour** of the system over time use an activity diagram.

Sequence Diagram

- The sequence diagram describes the flow of messages being passed from object to object.

The **purposes** of **interaction diagram** can be describes as:
- To capture **dynamic behavior** of a system.
- To describe the **message flow** in the system.
- To describe **structural organization** of the objects.
- To describe **interaction** among objects.

Sequence Diagram Elements

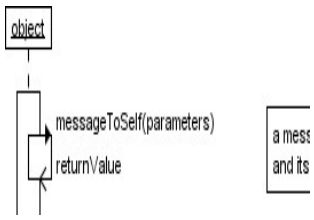
- **Class roles**, which represent **roles** that **objects** may play within the **interaction**.
- **Lifelines**, which represent the **existence** of an **object** over a period of time.
- **Activations**, which represent the **time** during which an object is performing an **operation**.
- The white rectangles on a **lifeline** are called **activations** and indicate that an object is ***responding to a message***. It starts when the message is received and ends when the object is done handling the message.
- **Messages**, which represent **communication** between **objects**.

Messages

- An **interaction** between **two objects** is performed as a **message** sent from **one object** to **another** (simple operation call, Signaling, RPC)
- If object obj_1 sends a message to another object obj_2 some link must exist between those two objects .
- A message is represented by an **arrow** between the life lines of two objects.
  - ***Self calls*** are also allowed
  - The time required by the receiver object to process the message is denoted by an ***activation-box***.
- A message is labeled at minimum with the **message name**.
  - **Arguments** and **control information** (conditions, iteration) may be included.

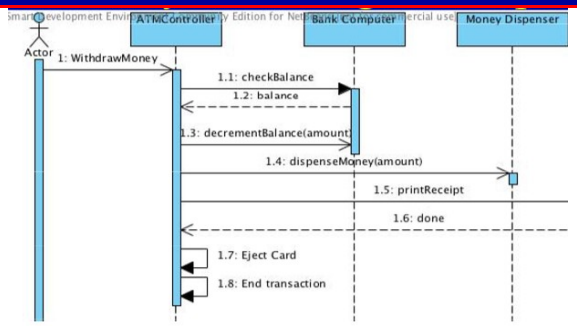
Message to Self

- A message that an object sends itself can be shown as follows:



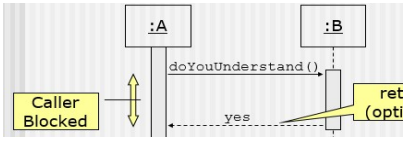
- Keep in mind that the purpose of a sequence diagram is to show the **interaction between objects**, so think twice about every self message you put on a diagram.

Asynchronous Messages



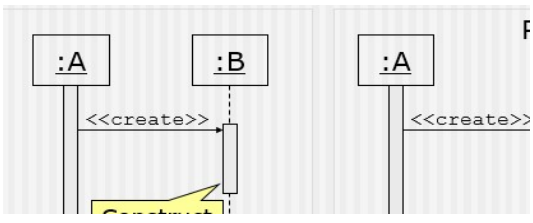
Return Values

- Optionally indicated using a **dashed arrow** with a label indicating the **return value**.
  - Don't model a return value when it is obvious what is being returned, e.g. getTotal()
  - Model a return value only when you need to *refer to it elsewhere*, e.g. as a **parameter passed** in another message.



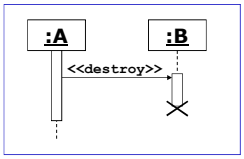
Object Creation

An object may create another object via a <<create>> message.

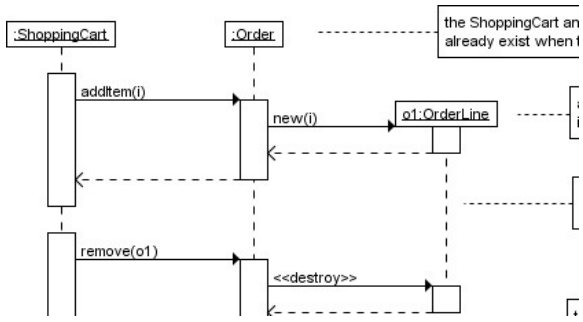


Object Destruction

- An object may destroy another object via a <<destroy>> message.
  - An object may destroy **itself**.
  - Avoid modeling object destruction unless **memory management** is critical.

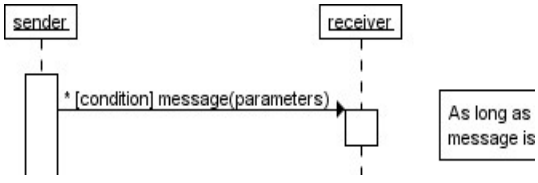


Creation and Destruction

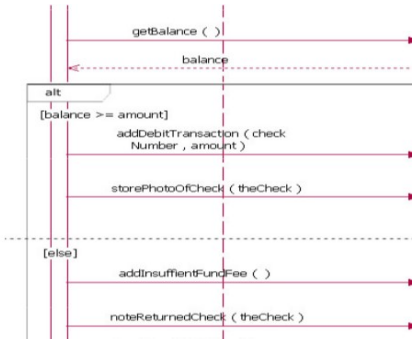


Repeated Interaction

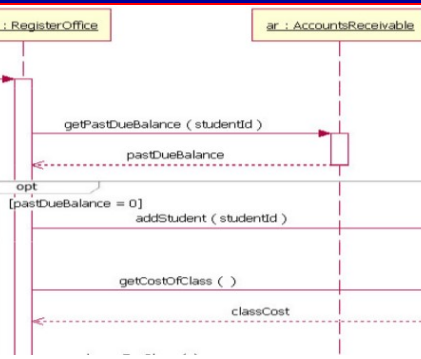
- When a message is prefixed with an **asterisk** (the '*'-symbol), it means that the message is *sent repeatedly*.
- A **guard** indicates the **condition** that determines whether or not the message should be sent (again). As long as the condition holds, the message is **repeated**.



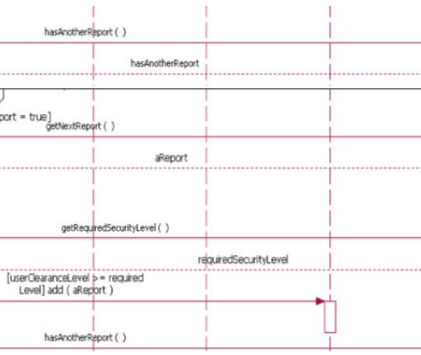
Components: alt/else



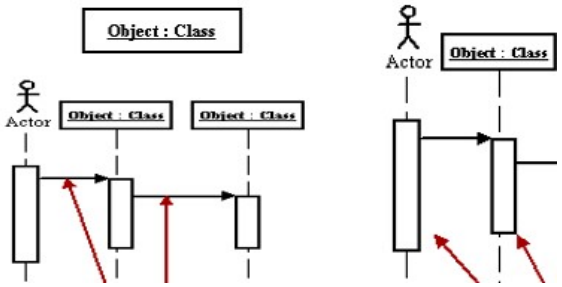
Components: option



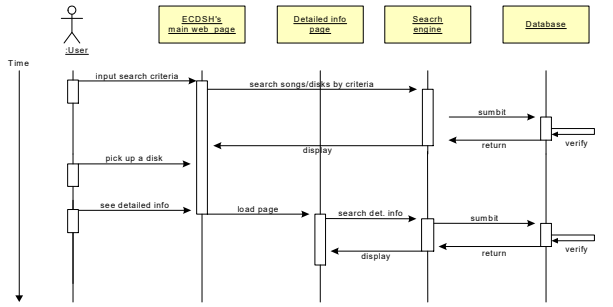
Components: loop



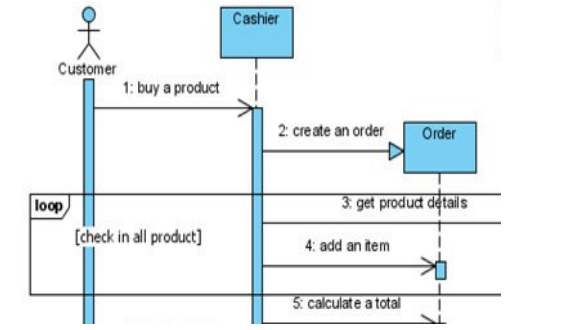
Sequence Diagram Notations



Sequence Diagram- *Search Engine*

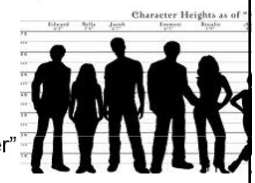


Sequence Diagram- *OMS*



Sequence Diagram- *Summary*

- Sequence diagrams model object interactions w
  emphasis on time ordering
- Method call lines
  - Must be horizontal!
  - Vertical height matters!
    "Lower equals Later"
  - Label the lines
- Lifeline – dotted vertical line
- Execution bar – bar around lifeline when code is
- Arrows
  - Synchronous call (you're waiting for a return value) –



Collaboration/ Communication Diagram

- Collaboration diagrams model the **interactions** between **objects**.
- This type of diagram is a **cross** between an **object diagram** and a **sequence diagram**.
- Unlike the Sequence diagram, which models the interaction in a column and row type format, the Collaboration diagram uses the **free-form arrangement** of **objects** as found in an Object diagram.
- This makes it easier to see all interations involving a particular object.
- Here in collaboration diagram the **method call sequence** is indicated by some **numbering technique** as shown below.
- The number indicates how the **methods** are called one after another.

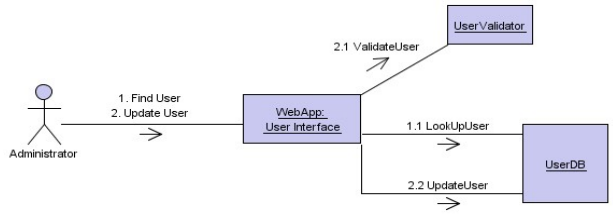
Communication Diagram

- The **method calls** are similar to that of a sequence diagram. But the difference is that the *sequence diagram does not describe the object organization* where as the **collaboration diagram** shows the **object organization.**
- If the **time sequence** is **important** then sequence diagram is used and if organization is required then collaboration diagram is used.

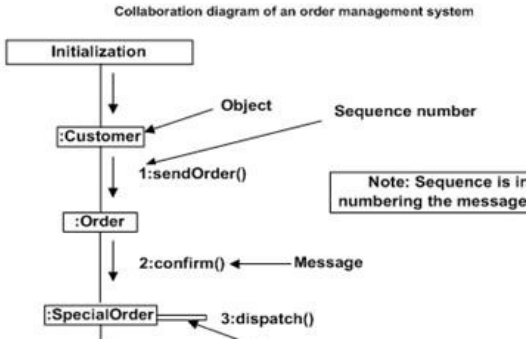
Communication Diagram Elements

- Collaboration Diagrams describe **interactions among classes** and **associations**. These interactions are modeled as *exchanges of messages between classes* through their **associations**. Collaboration diagrams are a type of interaction diagram. Collaboration diagrams contain the following elements.
- **Class roles**, which represent **roles that objects may play** within the interaction.
- **Association roles**, which represent **roles that links may play** within the interaction.
- **Message flows**, which represent **messages** sent between objects via links. Links transport or implement the delivery of the message.

Communication Diagram Example



Communication Diagram- *OMS*



Sequence vs. Communication

- Semantically both are the same
- Express different sides of the model
- Sequence diagram expresses time ordering
- Collaboration diagram is used to define class behavior

OBJECT ORIENTED SOFTWARE ENGINEERING

UNIT IV

Learning Objectives

- **Object Oriented Testing Techniques:**
  →Testing Terminology,
  →Types of test,
  → Automatic Tests,
  →Testing Strategies.
- **Agile Process**:
  →Agile Manifesto,
  → Agile Principles,
  →Introduction to Extreme Programming,
  →Scrum,
  → Lean processes.
- Case Studies.

TESTING

Object Oriented Testing Concepts

- Testing
- Verification
- Validation
- Debugging
- Certification
- Clean Room Software Engineering
- Error
- Fault
- Failure
- Testing Level
  - Unit testing
  - Integration Testing
  - System Testing

•Testing Techniques
  •Regression Test
•Testing Focuses
  •Operation test
  •Full-scale test
  •Stress test
  •Overload test
  •Negative test
  •Test based on requirements
  •Ergonomic tests
  •Testing of the user documentation
  •Acceptance testing

Object Oriented Testing Concepts.

- Stubs
- Drivers
- Test bed
- Equivalence set
- Equivalence partitioning
- Automatic Testing
  - Test data
- Test program

Software Testing

- Consumes at least half of the labor
- Process of testing software product
- Contribute to
  - Delivery of higher quality product
  - More satisfied users
  - Lower maintenance cost
  - More accurate and reliable results

Error

- People make errors.
- Typographical error
- Misreading of a specification
- Misunderstanding of functionality of a modı
- A good Synonym is Mistake.
- When people make mistakes while coding these mistakes **"bugs"**

Fault/Defect

Representation of an error

- DFD
- Hierarchy chart
- Source Code
- An error may lead to one or more faults

Fault of Omissions

- If certain specifications have not been programm

Fault of Commission

- If certain program behavior have not been specifi

Fault/Defect

- Defects generally fall into one of the followir categories
  - Wrong
  - Missing
  - Extra

Failure/Incident

Failure

- A particular fault may cause different failures, depending been exercised

Incident

- When a failure occurs, it may or may not readily apparent to t
- Incident is the symptom associated with a failure that alerts occurrence of a failure

Software Testing

- Software testing can be stated as the process of **validating** and **verifying** that a software program/application/product:
- Meets the **requirements** that guided its design and development works as expected;
- Software testing, depending on the testing method employed, can be implemented at any time in the development process. However, most of the **test effort** occurs after the requirements have been defined and the coding process has been completed.

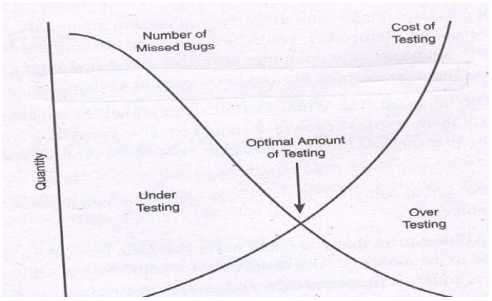
Software Testing

- A primary purpose of testing is to ***detect software failures*** so that defects may be discovered and corrected.
- Testing cannot establish that a product functions properly under all conditions but can only establish that it does not function properly under specific conditions

Testing takes creativity

- Testing often viewed as dirty work.
- To develop an effective test, one must have:
  - ✓ Detailed understanding of the system
  - ✓ Knowledge of the testing techniques
  - ✓ Skill to apply these techniques in an effective and efficient manner
- Testing is done best by **independent testers**
  - We often develop a certain mental attitude that the program should in a certain way when in fact it does not.

Testing Costs



Testing Objectives

- Executing a program with the goal of finding an *error*.
- To check if the system meets the requirements and be executed successfully in the planned environment.
- To check if the system is " Fit for purpose".
- To check if the system does what it is expected to do.

Tester Objectives

- Find bugs as **early** as possible and make sure they get fixed.
- To **understand** the application well.
- Study the functionality in **detail** to find where the bugs are likely to occur.
- Study the code to ensure that each and every line of code is tested.
- Create test cases in such a way that testing is done to uncover the **hidden bugs** and also ensure that the software is usable and reliable

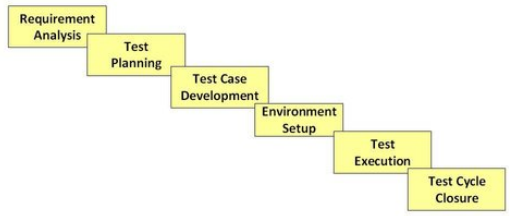
Verification and Validation

Verification - typically involves reviews and meeting to evaluate documents, plans, code, requirements, and specifications. This can be done with checklists, issues lists, and inspection meeting.

Validation - typically involves actual testing and takes place after verifications are completed.

In other words, validation is concerned with checking that the system will meet the customer's actual needs, while verification is concerned with whether the system is well-engineered, error-free, and so on. Verification will help to determine whether the software is of high quality, but it will not ensure that the system is useful.

Software Testing Life Cycle



Requirement Analysis

Activities

- Identify types of tests to be performed.
- Gather details about testing priorities and focus.
- Prepare Requirement Traceability Matrix (RTM).
- Identify test environment details where testing is supposed to be carried out.
- Automation feasibility analysis (if required).

Deliverables

- RTM
- Automation feasibility report. (if applicable)

Test Planning

Activities

- Preparation of test plan/strategy document for various types of testing
- Test tool selection
- Test effort estimation
- Resource planning and determining roles and responsibilities.
- Training requirement

Deliverables

- Test plan /strategy document.
- Effort estimation document.

Test Case Development

Activities

- Create test cases, automation scripts (if applicable)
- Review and baseline test cases and scripts
- Create test data (If Test Environment is available)

Deliverables

- Test cases/scripts
- Test data

Test Environment Setup

Activities

- Understand the required architecture, environment set-up and prepare hardware and software requirement list for the Test Environment.
- Setup test Environment and test data
- Perform smoke test on the build

Deliverables

- Environment ready with test data set up
- Smoke Test Results.

Test Execution

Activities

- Execute tests as per plan
- Document test results, and log defects for failed cases
- Map defects to test cases in RTM
- Retest the defect fixes
- Track the defects to closure

Deliverables

- Completed RTM with execution status
- Test cases updated with results
- Defect reports

Test Cycle Closure

Activities

- Evaluate cycle completion criteria based on Time. Test coverage, Cost. Software, Critical Business Objectives , Quality
- Prepare test metrics based on the above parameters.
- Document the learning out of the project
- Prepare Test closure report
- Qualitative and quantitative reporting of quality of the work product to the customer.
- Test result analysis to find out the defect distribution by type and severity.

Deliverables

- Test Closure report
- Test metrics

Testing Levels

- Unit testing
- Integration testing
- System testing
- Acceptance testing

Types of Testing

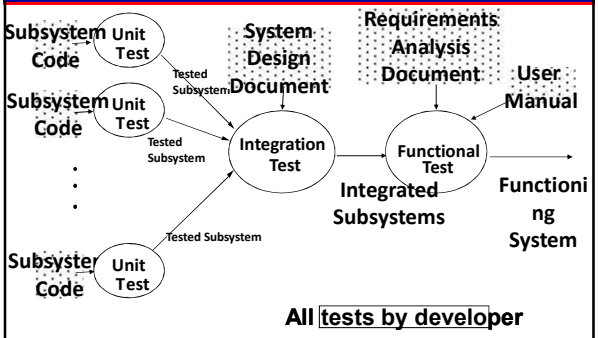
- Unit Testing:
  - Individual subsystem
  - Carried out by developers
  - Goal: Confirm that subsystems is correctly coded and carries out the intended functionality
- Integration Testing:
  - Groups of subsystems (collection of classes) and eventually the entire system
  - Carried out by developers
  - Goal: Test the interface among the subsystem

System Testing

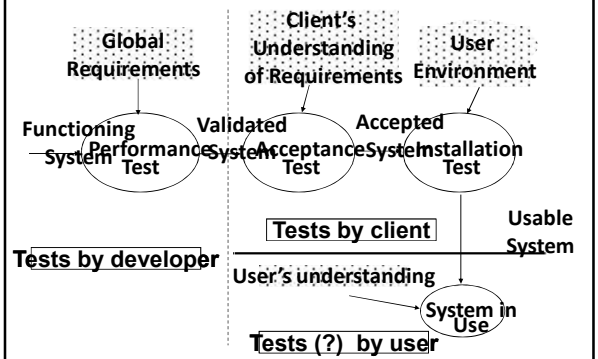
- System Testing:
  - The entire system
  - Carried out by developers
  - Goal: Determine if the system meets the requirements (functional and global)
- Acceptance Testing:
  - Evaluates the system delivered by developers
  - Carried out by the client. May involve executing typical transactions on site on a trial basis
  - Goal: Demonstrate that the system meets customer requirements and is ready to use

Implementation (Coding) and testing go hand in hand

Testing Activities



Testing Activities



Unit testing

- The most 'micro' scale of testing.
- Tests done on particular functions or code modules.
- Requires knowledge of the internal program design and code.
- Done by Programmers (not by testers).

Unit testing

Objectives	• To test the function of a program or unit of code such as a program or module • To test internal logic • To verify internal design • To test path & conditions coverage • To test exception conditions & error handling
When	• After modules are coded
Input	• Internal Application Design • Unit Test Plan
Output	• Unit Test Report

Unit testing

Who	•Developer
Methods	•White Box testing techniques
Tools	•Debug •Re-structure •Code Analyzers •Path/statement coverage tools
Education	•Testing Methodology •Effective use of tools

Incremental Integration Testing

- Continuous testing of an application as and when a new functionality is added.
- Application's functionality aspects are required to be independent enough to work separately before completion of development.
- Done by programmers or testers.

Integration Testing

Integration Testing

- Testing of **combined parts** of an application to determine their **functional correctness**.
- 'Parts' can be
  - ✓ code modules
  - ✓ individual applications
  - ✓ client/server applications on a network.

Integration Testing

Types of Integration Testing

- Big Bang testing
- Top Down Integration testing
- Bottom Up Integration testing

Integration Testing

Objectives	• To technically verify proper interfacing between modules, and within sub-systems
When	• After modules are unit tested
Input	• Internal & External Application Design • Integration Test Plan
Output	• Integration Test report

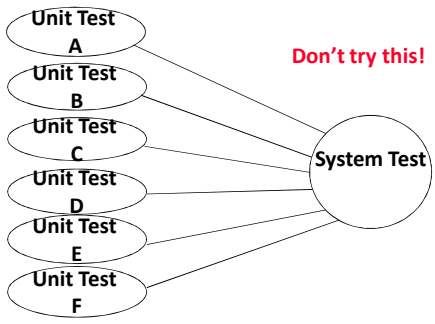
Integration Testing

Who	•Developers
Methods	•White and Black Box techniques •Problem / Configuration Management
Tools	•Debug •Re-structure •Code Analyzers
Education	•Testing Methodology •Effective use of tools

Integration Testing Strategy

- The entire system is viewed as a collection of subsystems (sets of classes) determined during the system and object design.
- The order in which the subsystems are selected for testing and integration determines the testing strategy
  - Big bang integration (Non incremental)
  - Bottom up integration
  - Top down integration
  - Sandwich testing
  - Variations of the above
- For the selection use the system decomposition from the System Design

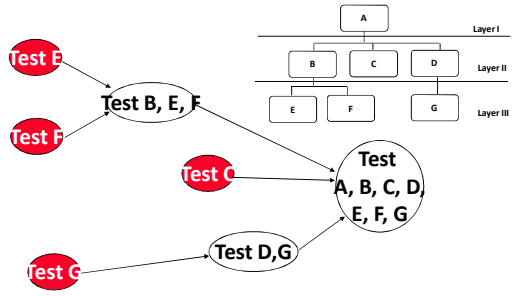
Integration Testing: Big-Bang Approac



Bottom-up Testing Strategy

- The subsystem in the lowest layer of the call hierarchy are tested individually
- Then the next subsystems are tested that call the previously tested subsystems
- This is done repeatedly until all subsystems are included in the testing

Bottom-up Integration



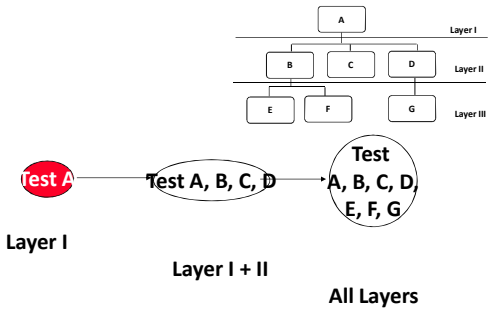
Pros and Cons of bottom up integration testin

- Bad for functionally decomposed systems:
  - Tests the most important subsystem (UI) last
- Useful for integrating the following systems
  - Object-oriented systems
  - real-time systems
  - systems with strict performance requirements

Top-down Testing Strategy

- Test the top layer or the controlling subsystem first
- Then combine all the subsystems that are called by the **tested subsystems** and test the resulting collection of subsystems
- Do this until all subsystems are incorporated into the test

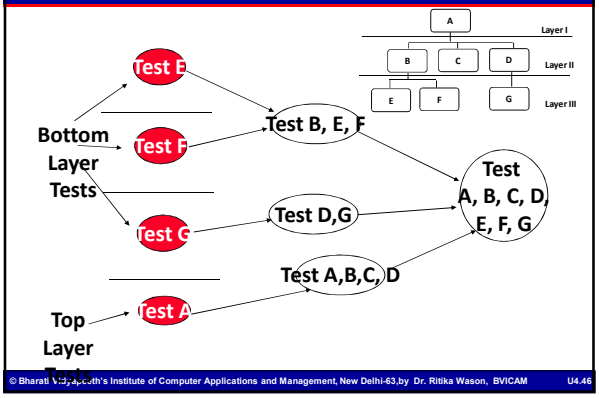
Top-down Integration Testing



Sandwich Testing Strategy

- *Combines top-down strategy with bottom-up strategy*
- *The system is view as having three layers*
  - A target layer in the middle
  - A layer above the target
  - A layer below the target
  - Testing converges at the target layer
- How do you select the target layer if there are more than 3 layers?
  - Heuristic: Try to minimize the number of stubs and drivers

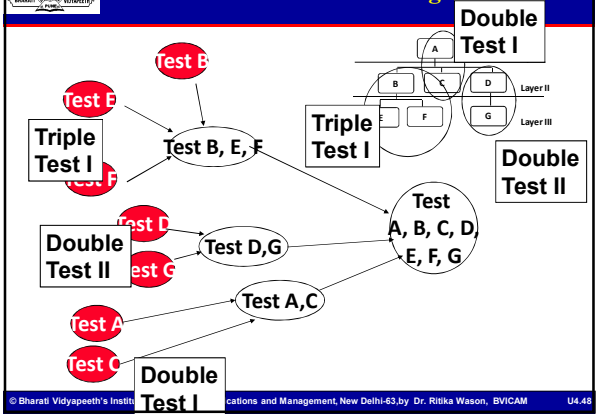
Sandwich Testing Strategy



Pros and Cons of Sandwich Testing

- Top and Bottom Layer Tests can be done in parallel
- Does not test the individual subsystems thoroughly before integration
- Solution: Modified sandwich testing strategy
- **Test in parallel:**
  - Middle layer with drivers and stubs
  - Top layer with stubs
  - Bottom layer with drivers
- **Test in parallel:**
  - Top layer accessing middle layer (top layer replaces drivers)
  - Bottom accessed by middle layer (bottom layer replaces

Modified Sandwich Testing Strategy



System Testing

- Functional Testing
- Structure Testing
- Performance Testing
- Acceptance Testing
- Installation Testing

Impact of requirements on system testing:

- The more explicit the requirements, the easier they are to test.
- Quality of use cases determines the ease of functional testing
- Quality of subsystem decomposition determines the ease of structure testing
- Quality of nonfunctional requirements and constraints determines the ease of performance tests:

Structure Testing

- *Essentially the same as white box testing.*
- Goal: Cover all paths in the system design
  - Exercise all input and output parameters of each component.
  - Exercise all components and all calls (each component is called at least once and every component is called by all possible callers.)
  - Use conditional and iteration testing as in unit testing.

Functional Testing

Essentially the same as black box testing

- Goal: Test functionality of system
- Test cases are designed from the requirements analysis document (better: user manual) and centered around requirements and key functions (use cases)
- The system is treated as black box.
- Unit test cases can be reused, but in end user oriented new test cases have to be developed as well.

Performance Testing

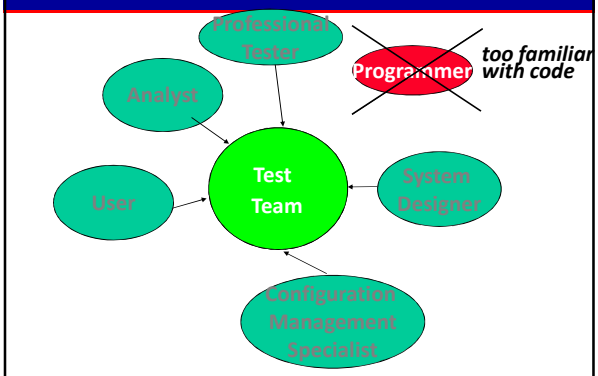
- Stress Testing
  - Stress limits of system (maximum # of users, peak demands, extended operation)
- Volume testing
  - Test what happens if large amounts of data are handled
- Configuration testing
  - Test the various software and hardware configurations
- Compatibility test
  - Test backward compatibility with existing systems
- Security testing
  - Try to violate security requirements
- Timing testing
  - Evaluate response times and time to perform a function
- Environmental test
  - Test tolerances for heat, humidity, motion, portability
- Quality testing
  - Test reliability, maintain- ability & availability of the system
- Recovery testing
  - Tests system's response to presence of errors or loss of data.
- Human factors testing
  - Tests user interface with user

Acceptance Testing

- Goal: Demonstrate system is ready for operational use
  - Choice of tests is made by client/sponsor
  - Many tests can be taken from integration testing
  - Acceptance test is performed by the client, not by the developer.
- Majority of all bugs in software is typically found by the client after the system is in use, not by the developers or testers. Therefore two kinds of additional tests:

- *Alpha test:*
  - Sponsor uses the software at the *developer's site.*
  - Software used in a controlled setting, with the developer always ready to fix bugs.
- *Beta test:*
  - Conducted at *sponsor's site* (developer is not present)
  - Software gets a realistic workout in target environ-ment
  - Potential customer might get discouraged

Test Team



System Testing

Objectives	• To verify that the system components perform control functions • To perform inter-system test • To demonstrate that the system performs both functionally and operationally as specified • To perform appropriate types of tests relating to Transaction Flow, Installation, Reliability etc.
When	• After Integration Testing
Input	• Detailed Requirements & External Application Design • Master Test Plan • System Test Plan
Output	• System Test Report

System Testing

Who	•Development Team and Users
Methods	•Problem / Configuration Management
Tools	•Recommended set of tools
Education	•Testing Methodology •Effective use of tools

TESTING METHODOLOGIES AND TYPES

Testing Methodologies

Black box testing

- No knowledge of internal design or code required.
- Tests are based on requirements and functionality

White box testing

- Knowledge of the internal program design and code required.
- Tests are based on coverage of code statements, branches, paths, conditions.

Testing Methodologies

Black box / Functional testing

Based on requirements and functionality

Not based on any knowledge of internal design or code

Covers all combined parts of a system

Tests are data driven

Black-box Testing

- Focus: I/O behavior. If for any given input, we can predict the output, then the module passes the test.
  - Almost always impossible to generate all possible inputs ("test cases")
- Goal: Reduce number of test cases by equivalence partitioning:
  - Divide input conditions into equivalence classes
  - Choose test cases for each equivalence class. (Example: If an object is supposed to accept a negative number, testing one negative number is enough)

White box testing / Structural testing

Based on knowledge of internal logic of an application's code

Based on coverage of code statements, branches, paths, conditions

Tests are logic driven

White-box Testing

- Statement Testing (Algebraic Testing): Test single statements (Choice of operators in polynomials, etc)
- Loop Testing:
  - Cause execution of the loop to be skipped completely. (Exception: Repeat loops)
  - Loop to be executed exactly once
  - Loop to be executed more than once
- Path testing:
  - Make sure all paths in the program are executed
  - Branch Testing (Conditional Testing): Make sure that each possible outcome from a condition is tested at least once

White Box - Testing Technique

- All independent paths within a module have been exercised at least once
- Exercise all logical decisions on their *true* and sides
- Execute all loops at their boundaries and within their operational bounds
- Exercise internal data structures to ensure their validity

Loop Testing

This white box technique focuses on the validity of loop constructs.

Different classes of loops can be defined

- simple loops
- nested loops

Other White Box Techniques

Statement Coverage – execute all statements at least once

Decision Coverage – execute each decision direction at least once

Condition Coverage – execute each decision with all possible outcomes at least once

Decision / Condition coverage – execute all possible combinations of condition outcomes in each decision.

Multiple condition Coverage – Invokes each point of entry at least once.

Comparison White & Black-box Testing

- White-box Testing:
  - Potentially infinite number of paths have to be tested
  - White-box testing often tests what is done, instead of what should be done
  - Cannot detect missing use cases
- Black-box Testing:
  - Potential combinatorial explosion of test cases (valid & invalid data)
  - Often not clear whether the selected test cases uncover a particular error
  - Does not discover extraneous use cases ("features")
- Both types of testing are needed
- White-box testing and black box testing are the extreme ends of a testing continuum.
- Any choice of test case lies in between and depends on the following:
  - Number of possible logical paths
  - Nature of input data
  - Amount of computation
  - Complexity of algorithms and data structures

Testing Techniques

Functional testing

- Black box type testing geared to functional requirements of an application.
- Done by testers.

System testing

- Black box type testing that is based on overall requirements specifications; covering all combined parts of the system.

End-to-end testing

- Similar to system testing; involves testing of a complete application environment in a situation that copies real-world use.

Testing Techniques

Regression testing

- Re-testing after fixes or modifications of the software or its environment.

Acceptance testing

- Final testing based on specifications of the end-user or customer

Load testing

- Testing an application under heavy loads.
- Eg. Testing of a web site under a range of loads to determine, when the system response time degraded or fails.

Testing Techniques

Stress Testing

- Testing under unusually heavy loads, heavy repetition of certain actions or inputs, input of large numerical values, large complex queries to a database etc.
- Term often used interchangeably with 'load' and 'performance' testing.

Performance testing

- Testing how well an application complies to performance requirements.

Testing Techniques

Install/uninstall testing

- Testing of full, partial or upgrade install/uninstall process.

Recovery testing

- Testing how well a system recovers from crashes, HW failures or other problems.

Compatibility testing

- Testing how well software performs in a particular HW/SW/OS/NW environment.

Testing Techniques

Exploratory testing / ad-hoc testing

- Informal SW test **that is not based on formal test plans** or test cases; testers will be learning the SW in totality as they test it.

Comparison testing

- Comparing SW **strengths and weakness to competing products**.

Test Plan

Objectives

- To create a set of testing tasks.
- Assign resources to each testing task.
- Estimate completion time for each testing task.
- Document testing standards.

Test Cases

Test case is defined as

- A set of **test inputs**, **execution conditions** and expected results, developed for a particular objective.
- Documentation specifying inputs, predicted results and a set of execution conditions for a test item.

Test Cases

Contents

- Test plan reference id
- Test case
- Test condition
- Expected behavior

Good Test Cases

Find Defects

Have high probability of finding a new defect.

Unambiguous tangible result that can be inspected.

visible to requirements or design documents

Execution and tracking can be automated

Do not mislead

Feasible

Defect Log

- Defect ID number
- Descriptive defect name and type
- Source of defect – test case or other source
- Defect strictness
- Defect Priority
- Defect status (e.g. New, open, fixed, closed, reopen, reject)

Defect Log

7. Date and time tracking for either the most recent status change, or for each change in the status.
8. Detailed description, including the steps necessary to reproduce the defect.
9. Component or program where defect was found
10. Screen prints, logs, etc. that will aid the developer in resolution process.
11. Stage of origination.
12. Person assigned to research and/or corrects the defect.

Test Metrics

User Participation = User Participation test time **Vs.** Total test time.

Path Tested = Number of path tested **Vs.** Total number of paths.

Acceptance criteria tested = Acceptance criteria verified **Vs.** Total acceptance criteria.

Test Metrics

Test cost = Test cost **Vs.** Total system cost.

Cost to locate defect = Test cost / No. of defects located in the testing.

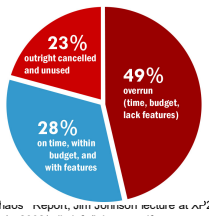
Detected production defect = No. of defects detected in production / Application system size.

Test Automation = Cost of manual test effort / Total test cost.

Extreme Programming and Scrum - Getting Started with Agile Software Development

The Problem: The Chaos Report

- Started in 1994, studied over 35,000 application development projects
- In 2000:



Source: Standish "Chaos" Report, Jim Johnson lecture at XP2002 conference, http://www.xp2003.org/xp2002/talksinfo/johnson.pdf

What is Needed: DuPont Study

- 25% of features are needed
- 75% of features are "nice to have"

Source: Jim Johnson lecture at XP2003 conference, http://www.xp2003.org/xp2002/talkinfo/johnson.pdf

Lifecycle Costs

- Up to 80% of software lifecycle cost, the total cost of ownership (TCO), is in *maintenance,* not first development
- Focusing on "abilities" is critical to ROI:
  - maintainability, extensibility, adaptability, scalability, and most importantly *understandability* (usability, readability, testability)

Agile Methods

- **Extreme Programming** (XP) (Kent Beck, Ward Cunningham, Ron Jeffries)
- **Scrum** (Jeff Sutherland, Mike Beedle, Ken Schwaber)
- **DSDM** – Dynamic Systems Development Method (Community owned)
- **Crystal** (Alistair Cockburn)
- **ASD** – Adaptive Software Development (Jim Highsmith)
- **XBreed** (Mike Beedle)

All Agile Methods

- Maximize value by minimizing anything that does not directly contribute to product development and delivery of customer value
- Respond to change by inspecting and adapting
- Stress evolutionary, incremental development
- Build on success, not hope

We've Seen It Before

- Lean Manufacturing (1990, Toyota)
- Agile Manufacturing
- Just-in-time JIT
- Common goals include:
  - Reduce Cycle Time
  - Maximize Quality
  - Reduce Costs

Lean

- Lean means ***prioritize and optimize everything to deliver value to the customer***
- One common technique: ***Postpone decisions until the last responsible moment***. Live with uncertainty but define, communicate, and manage it
  Lean Manufacturing and the Toyota Production System, SAE International
- Lean Software Development: An Agile Toolkit, Mary & Tom Poppendieck, 2003

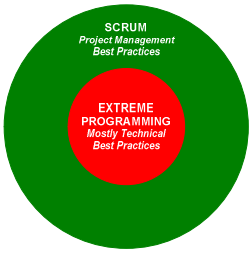
Scrum

- Term in rugby to get an out-of-play ball back into play
- Term used in Japan in 1987 to describe hyper-productive development
- Used by Ken Schwaber and Mike Beedle to describe their Agile methodology

Extreme Programming

- A collection of best practices – each done to the "extreme"
- Sounds extreme, but very disciplined
- Created by Kent Beck, Ward Cunningham, Ron Jeffries

Scrum with Extreme Programming



Scrum works well as a *wrapper* around Extreme Programming

Agile Independence

- Not created by any single company, but by a group of software industry experts to find "better ways of developing software by doing it and helping others do it."*
- Agile Principles:
  - highest priority is customer satisfaction
  - welcomes changing requirements
  - frequently deliver working software
  - advocates close collaboration and rapid feedback
  - reinforces "inspect and adapt"

* www.agilealliance.org

Key PM Difference

Defined/ Predictive Project Management
vs.
Empirical Project Management

Defined PM

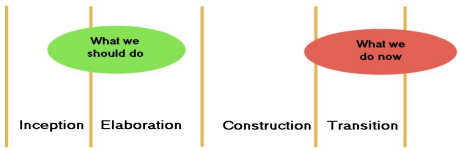
- ***Assumes we can predict*** how the project will unfold – ***assumes very little uncertainty***
- Time to complete and costs predictable
- Uses work breakdown structure
- Manages to a static plan
- Primary participants: development team
- Success; ***On Time & On Budget***

Empirical PM

- Software and systems construction ***is a discovery process*** – ***manages uncertainty***
- Focuses on value/cost tradeoffs
- Plan is volatile; use discoveries to reprioritize and adjust
- Primary participants: project steering team
- Success; ***Delivering good value in reasonable time***

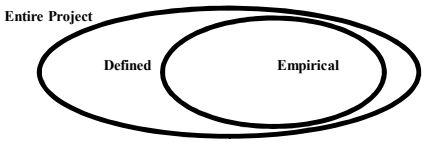
PM End-Game

- Almost all projects eventually revert to empirical PM



Both Are Needed

- Defined PM works because many things in a project *are* deterministic.
- Defined model provides constraints:
  - "deliver not the best solution, but the best we can afford"



Empirical PM Strategy

- Early estimates of cost and value, tied to business processes
- Deliver subsets of functionality prioritized by business value
- Reassess and re-plan to fit resources, schedule, and discoveries

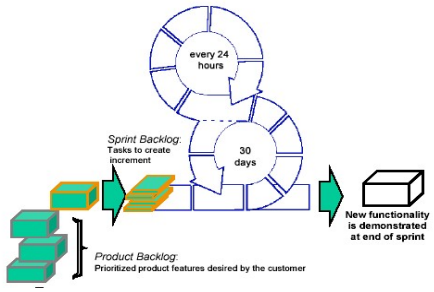
Agile PM Concepts

- Software construction is a discovery process
- Not the best solution; the affordable solution
- Invent successful outcomes

Scrum Overview

- Empirical management and control process for projects and products
- Widely used since 1990's
- Wraps existing engineering practices
- Manages noise, allows overhead to wither
- Simple, common sense
- Delivers business functionality in 30 days
- Scalable

Scrum Scheduling and Tracking



Scrum Roles

- Product Owner
- Team
- Scrum Master

Scrum Roles – Product Owner

- Single person who owns, maintains, prioritizes Product Backlog
- Empowered to make decisions for customers and users
- Responsible for vision, ROI, and releases of product
- Attends Sprint planning and Sprint review meetings

Scrum Roles - Team

- Self-organizing, cross-functional, no formal roles
- Seven plus or minus two people
- Best experts available
- Cost and commit to work, and responsible for delivering
- Full autonomy and authority to deliver during Sprint

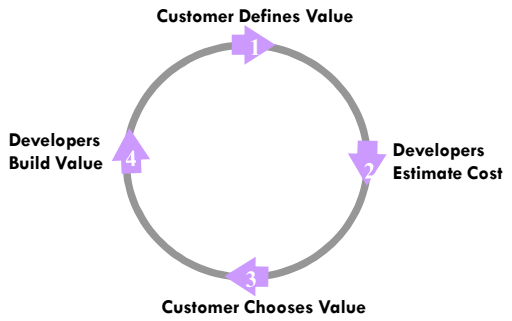
Scrum Roles – Scrum Master

- Project manager, Coach, and/or Player-Coach
- Responsible for process and maximizing team productivity
- Sets up and conducts meetings
  - Sprint Planning
  - Daily "Scrum"
  - Sprint Release

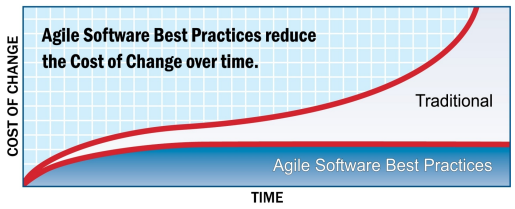
XP Definitions

- Kent Beck's idea of turning the knobs on all the best practices up to 10.
- Optimizing the "Circle of Life" by hitting the sweet-spot of practices that self-reinforce and become more than the sum of the parts (synergize).

XP Circle of Life



Cost-of-Change Curves



Flattening cost of change curve is both *enabled by* and *exploited by* Extreme Programming (XP)

The Four XP Values

- **Simplicity**
  - Simplest thing that could possibly work
  - YAGNI: you aren't going to need it
- **Communication**
  - Developers
  - Users
  - Customers
  - Testers
  - Code
- **Feedback**
  - Testing
  - Experimenting
  - Delivering
- **Courage**
  - Trust
  - History

The Four XP Variables

- **Quality**
  - **Internal high, fixed**
- **Cost**
  - **People-Time**
  - **Mythical Man-Month (F. Brooks)**
- **Schedule**
  - **Fixed-length, short iterations**
- **Scope**
  - **Negotiable**

Twelve XP Practices

1. **Planning Game**
2. **Short Releases**
3. **Simple Design**
4. **Testing**
5. **Refactoring**
6. **Pair Programming**
7. **Collective Ownership**
8. **Continuous Integration**
9. **On-site Customer**
10. **Sustainable Pace**
11. **Metaphor**
12. **Coding Standards**

1. Planning Game

- Release Planning: Define and estimate higher-level features down to about 5-10 days effort each. Customer lays features in fixed-length iteration schedule.
- Iteration Planning: Same, but to 3 or less days effort & detailed story cards within next iteration.
- Simple to steer project towards success.

2. Short Releases

- Deliver business value early and often
- Do not slip iteration release dates
  - adjust scope within an iteration, never time or quality
- Small, stable teams are predictable in short time-frames

3. Simple Design

- XP Mantra: "The simplest thing that could *possibly* work".
- Meet current, minimum business requirements only. Avoid anticipatory design.
- YAGNI – You Aren't Going to Need It

4. Testing

- Automated unit tests for every entity.
- Automated acceptance tests for every story / requirement.
- All unit tests pass 100% before checking in a feature.
- Test-First, in small increments:
  1. Write the test
  2. Prove it fails (red-bar)
  3. Code until it passes (green-bar)

5. Refactoring

- Refactoring: changing internal structure without changing external behavior
- Remove duplication. “Once and Only Once”, “Three strikes and your out”.
- Leaves code in simplest form.
- When change is hard, refactor to allow change to be easy, testing as you go, then add change.

6. Pair Programming

- Two heads are better than one, especially in an open lab environment (colocation)
- Earliest possible code inspections
- Earliest possible brainstorming
- Better quality at lower cost
- Driver/Navigator
- Peer pressure reinforces discipline

7. Collective Ownership

- Interchangeable programmers
- Team can go at full speed
- Can change anything, anytime, without delay

8. Continuous Integration

- Avoids "versionitis" by keeping all the programmers on the same page
- Integration problems smaller, taken one at a time
- Eliminates traditional, high-risk integration phase

9. On-site Customer

- Customer/User liaisons are team-members
- Available for priorities, clarifications, to answer detailed questions
- Reduces programmer assumptions about business value
- Shows stakeholders what they pay for, and why

10. Sustainable Pace

- Tired programmers make more mistakes
- Better to stay fresh, healthy, positive, and effective
- XP is for the average programmer, for the long run

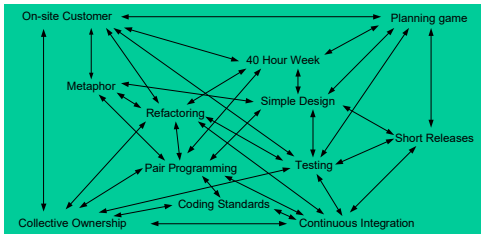
11. Metaphor

- Use a "system of names"
- Use a common system description
- Helps communicate with customers, users, stakeholders, and programmers

12. Coding Standards

- All programmers write the same way
- Rules for how things communicate with each other
- Guidelines for what and how to document

XP Practices Support Each Other



Source: Beck, Extreme Programming Explained: Embrace Change, 1999

Practices to Start With

- Talking to, instead of about, people, in their language, considering their perspective
  - Customer, developer, mgmt., Q/A, user, finance, marketing, sponsor
- Frequent Integration (Config. Mgmt., Check-in > daily)
- Testing (Unit, Integration, System, Feature)
- Release Management (build-box, sandboxes, labeled releases, migrations)

See www.balancedagility.com

How to Explore

Web

- Agile Alliance: www.agilealliance.org
- Scrum: www.controlchaos.com
- Don Well's XP Introduction: *Extreme Programming: A Gentle Introduction* www.extremeprogramming.org

How Not to Get Started

1. Read some
2. Discuss some
3. Start an approach without advice from those with previous experience
4. Draw conclusions from experience

- Can work this way, but its risky
- Often fails to define and leverage success criteria. Often unrealistic expectations.
- Inexperience decreases chances of success

How Best to Get Started

- Get help from experienced people for:
  - Readiness assessments
  - Approach selection
    - ✓Pilot / skunkworks vs. changing existing process
    - ✓Mission-critical vs. stand-alone
    - ✓Selective best practices vs. complementary set vs. all best practices
  - Measurement and success criteria
  - Identifying and delivering targeted training, mentoring, coaching, project management / stewardship

Agile Best Practice Adaptations

- How long should iterations and releases be?
- How does development work with QA?
- How do our stakeholders work with multiple customers?
- How should our teams be structured?
- How do we work with regulatory agencies?
- How does this work with legacy systems?
- How does this work with Use Cases and RUP?
- How do we ensure architectural vision and usage.

Agile Summary

- Agile = try, inspect, adapt, repeat
- Highly focused, empowered teams
- Collaborate with all stakeholders
- Optimize and automate feedback
- Deliver real value early and often
- Use feedback to evaluate, ruthlessly prioritize, and re-plan
- Delivers high quality, ensures flexibility
- Evaluate business value of everything

Agile Future

- Agile in most dev. orgs, in few IT orgs.
- Agile is here to stay, past early adopters, into early majority
- "Agile" is loosing meaning
- XP is developer-focused, now Q/A friendly, needs to become customer/user friendly
- Scrum is still "pure", but there are now tools… CMM and RUP were "pure" to start…
- All camps need to sell business value, in business terms, financial terms, risk terms