

## CS6502- OBJECT ORIENTED ANALYSIS AND DESIGN

### UNIT – I

Introduction to OOAD – Unified Process - UML diagrams – Use Case – Class Diagrams– Interaction Diagrams – State Diagrams – Activity Diagrams – Package, component and Deployment Diagrams.

---

#### **1. Introduction to OOAD**

##### **1.1 What is analysis and design?**

Analysis emphasizes an investigation of the problem and requirements, rather than a solution. For example, if a new computerized library information system is desired, how it will be used.

Design emphasizes a conceptual solution that fulfills the requirements, rather than its implementation. For example, a description of a database schema and software objects.

##### **1.2 What is object oriented analysis and design? (April/May 2011, 2017)**

During object oriented analysis, there is an emphasis on finding and describing the objects or concepts in the problem domain. For example, in the case of the library information system, some of the concepts include book, library and patron.

During object oriented design, there is an emphasis on defining software objects and how they collaborate to fulfill the requirements. For example, in the library system, a book software object may have a title attribute and a get chapter method

##### **1.3 Define Object. (Nov/Dec 2009)**

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

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

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

##### **1.4 Class**

A class represents a collection of objects having same characteristic properties that exhibit common behavior. It gives the blueprint or description of the objects that can be created from it.

Creation of an object as a member of a class is called instantiation. Thus, object is an instance of a class.

The constituents of a class are:

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

### 1.5 Encapsulation

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

### 1.6 Data Hiding

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

### 1.7 Message Passing

Any application requires a number of objects interacting in a harmonious manner. Objects in a system may communicate with each other using message passing. The features of message passing are:

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

### 1.8 Inheritance

Inheritance is the mechanism that permits new classes to be created out of existing classes by extending and refining its capabilities. The existing classes are called the base classes/parent classes/super-classes, and the new classes are called the derived classes/child classes/subclasses.

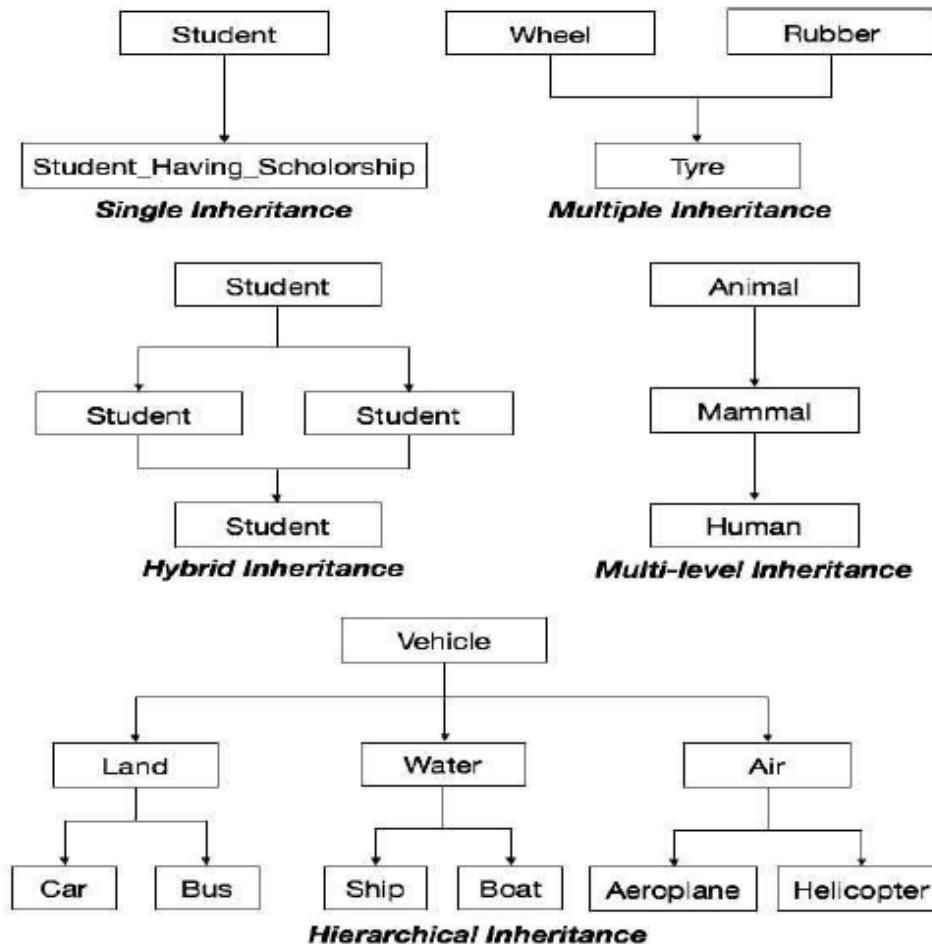
The subclass can inherit or derive the attributes and methods of the super-class(es) provided that the super-class allows so. Inheritance defines an “is – a” relationship.

#### Types of Inheritance:

- **Single Inheritance** : A subclass derives from a single super-class.

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

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



### 1.9 Polymorphism

Polymorphism is originally a Greek word that means the ability to take multiple forms. In object-oriented paradigm, polymorphism implies using operations in different ways, depending

upon the instance they are operating upon. Polymorphism allows objects with different internal structures to have a common external interface. Polymorphism is particularly effective while implementing inheritance.

### 1.10 Generalization and Specialization

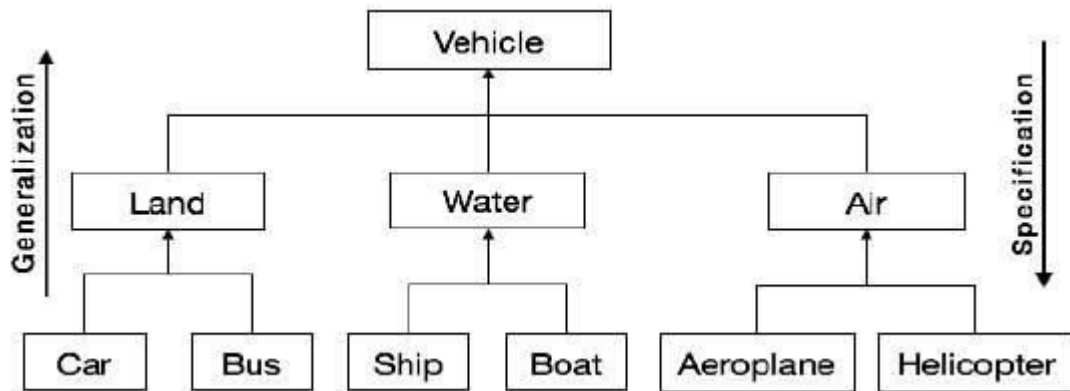
Generalization and specialization represent a hierarchy of relationships between classes, where subclasses inherit from super-classes.

#### Generalization:

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

#### Specialization:

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



### 1.11 Links and Association

#### Link:

A link represents a connection through which an object collaborates with other objects. Through a link, one object may invoke the methods or navigate through another object. A link depicts the relationship between two or more objects.

#### Association:

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

### ***Degree of an Association***

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

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

### ***Cardinality Ratios of Associations***

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

- **One-to-One** : A single object of class A is associated with a single object of class B.
- **One-to-Many** : A single object of class A is associated with many objects of class B.
- **Many-to-Many** : An object of class A may be associated with many objects of class B and conversely an object of class B may be associated with many objects of class A.

## **1.12 Aggregation or Composition**

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

Aggregation may denote:

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

## **2. Unified Process**

### **What do you mean by Unified process in OOAD? (Nov/Dec 2011, May/June 2016, April/May 2017)**

Unified Process is a popular iterative process for projects using object oriented analysis and design. UP make use of best practices such as iterative life cycle and risk-driver development to provide a well documents process description.

The Unified Process supports the following

1. Evolution of project plans, requirements and software architecture with well defined synchronization points
2. Risk management
3. Evolution of system capabilities through demonstrations of increasing functionality



It emphasizes the difference between *engineering* and *production*.

- **Engineering Stage**

- Driven by less predictable but smaller teams, focusing on design and synthesis activities

- **Production Stage**

- Driven by more predictable but larger teams, focusing on construction, test and deployment activities

## **Briefly explain the Iterative and Evolutionary Development**

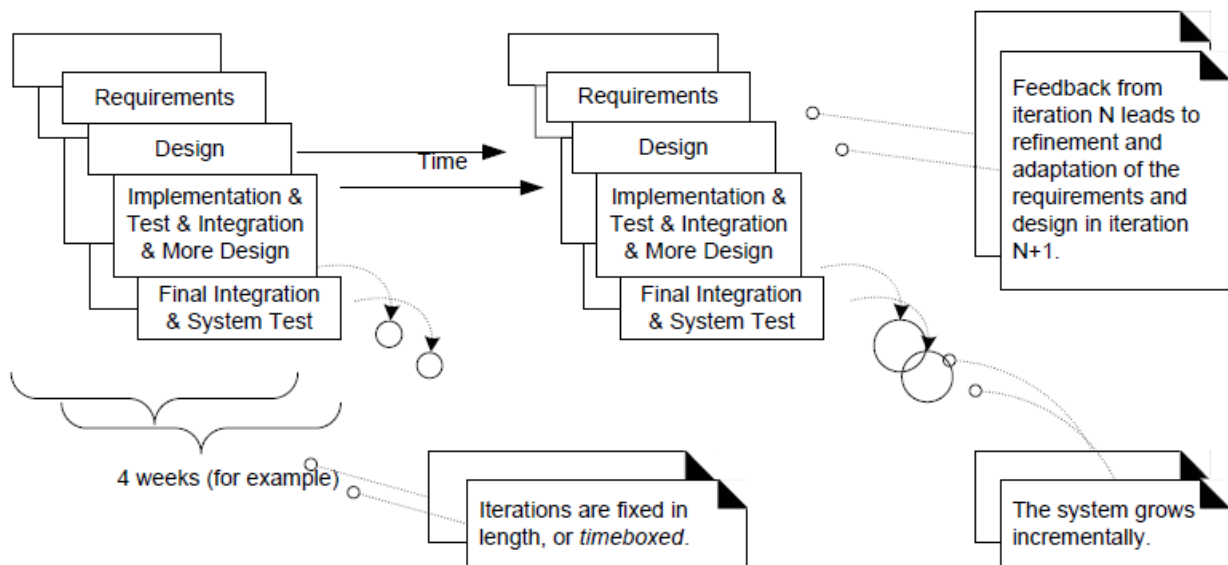
### **2.1 Iterative and Evolutionary Development**

- In this lifecycle approach, development is organized into a series of short, fixed-length (for example, three-week) mini-projects called **iterations**; the outcome of each is a tested, integrated, and executable partial system.

- Each iteration includes its own requirements analysis, design, implementation, and testing activities.

- The system grows incrementally over time, iteration by iteration, and thus this approach is also known as iterative and incremental development (see Fig1 given below).

- Because feedback and adaptation evolve the specifications and design, it is also known as iterative and evolutionary development.



**Fig1. Iterative and evolutionary development.**

**Briefly explain the different phases of unified process.(April/May 2011,Nov/Dec 2011, May/June 2012, Nov/Dec 2011,Nov/Dec 2015, May/Jun 2016, April/May 2017).**

## 2.2 Phases in the Unified Process

The two stages of the Unified Process are decomposed into four distinct phases

### *Engineering stage*

1. **Inception** - approximate vision, business case, scope, vague estimates.
2. **Elaboration** - refined vision, iterative implementation of the core architecture, resolution of high risks, identification of most requirements and scope, more realistic estimates.

### *Production phase*

3. **Construction** - iterative implementation of the remaining lower risk and easier elements, and preparation for deployment.
4. **Transition** - beta tests, deployment.

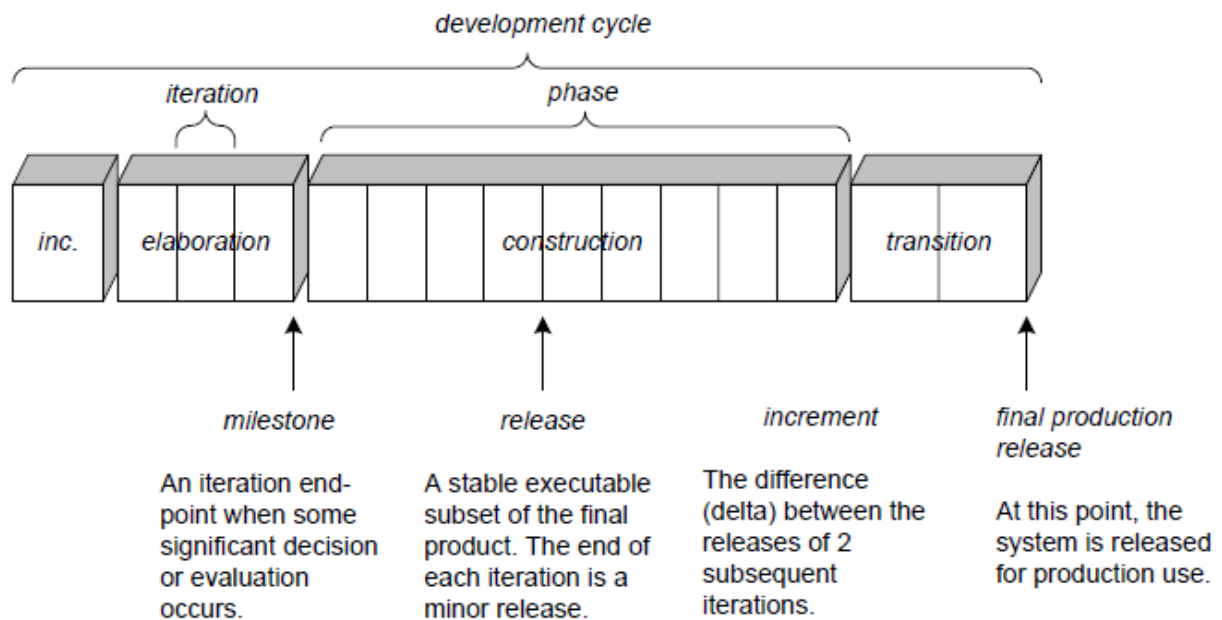


Fig2. Schedule-oriented terms in the UP.

## 2.3 The UP Disciplines (was Workflows)

The UP describes work activities, such as writing a use case, within **disciplines** (originally called **workflows**).

In the UP, an **artifact** is the general term for any work product: code, Web graphics, database schema, text documents, diagrams, models, and so on. There are several disciplines in the UP; focuses on some artifacts in the following three:

- **Business Modeling**— When developing a single application, this includes domain object modeling. When engaged in large-scale business analysis or business process reengineering, this includes dynamic modeling of the business processes across the entire enterprise.

- **Requirements**—Requirements analysis for an application, such as writing uses cases and identifying non-functional requirements.
- **Design**—All aspects of design, including the overall architecture, objects, databases, networking, and the like.

### *Disciplines and Phases*

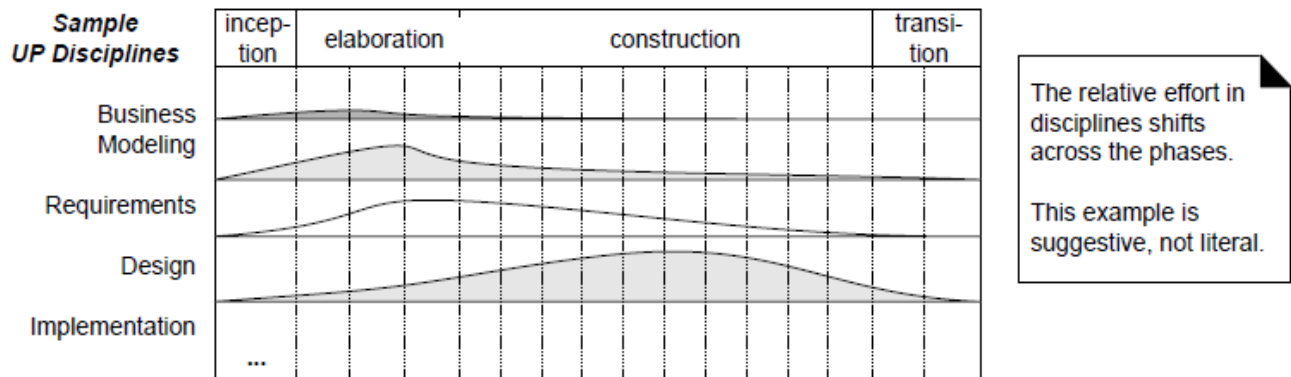


Fig3. Disciplines and phases

As illustrated in Fig3, during one iteration work goes on in most or all disciplines. However, the relative effort across these disciplines changes over time.

- Early iterations naturally tend to apply greater relative emphasis to requirements and design, and later ones less so, as the requirements and core design stabilize through a process of feedback and adaptation.

Relating this to the UP phases,

- In elaboration, the iterations tend to have a relatively high level of requirements and design work, although definitely some implementation as well.
- During construction, the emphasis is heavier on implementation and lighter on requirements analysis.

### **2.4 The Agile UP**

- Methodologists speak of processes as heavy vs. light, and predictive vs. adaptive.
- A **heavy process** is a pejorative term meant to suggest one with the following qualities:
  - many artifacts created in a bureaucratic atmosphere
  - rigidity and control
  - elaborate, long-term, detailed planning
  - predictive rather than adaptive



- A **predictive process** is one that attempts to plan and predict the activities and resource (people) allocations in detail over a relatively long time span, such as the majority of a project.
- In contrast, an **adaptive process** is one that accepts change as an inevitable driver and encourages flexible adaptation; they usually have an iterative lifecycle.
- An **agile process** implies a light and adaptive process, nimble in response to changing needs.
- The UP was not meant by its authors to be either heavy or predictive, although its large optional set of activities and artifacts have understandably led to that of an agile process—**agile UP**.
- A detailed plan (called the **Iteration Plan**) only plans with greater detail one iteration in advance.
- Detailed planning is done adaptively from iteration to iteration.
- Planning iterative projects, and the justification for this approach.
- The case study emphasizes a relatively small number of artifacts, and iterative development, in the spirit of an agile UP.

### **What are the benefits of Iterative development?**

#### ***2.5 Benefits of Iterative Development***

Benefits of iterative development include:

- Early rather than late mitigation of high risks (technical, requirements, objectives, usability, and so forth)
- Early visible progress
- Early feedback, user engagement, and adaptation, leading to a refined system that more closely meets the real needs of the stakeholders
- Managed complexity; the team is not overwhelmed by "analysis paralysis" or very long and complex steps
- The learning within an iteration can be methodically used to improve the development process itself, iteration by iteration

### **3. Unified Modelling Language (UML)**

#### **What is UML? (May/June 2012, April/May 2017)**

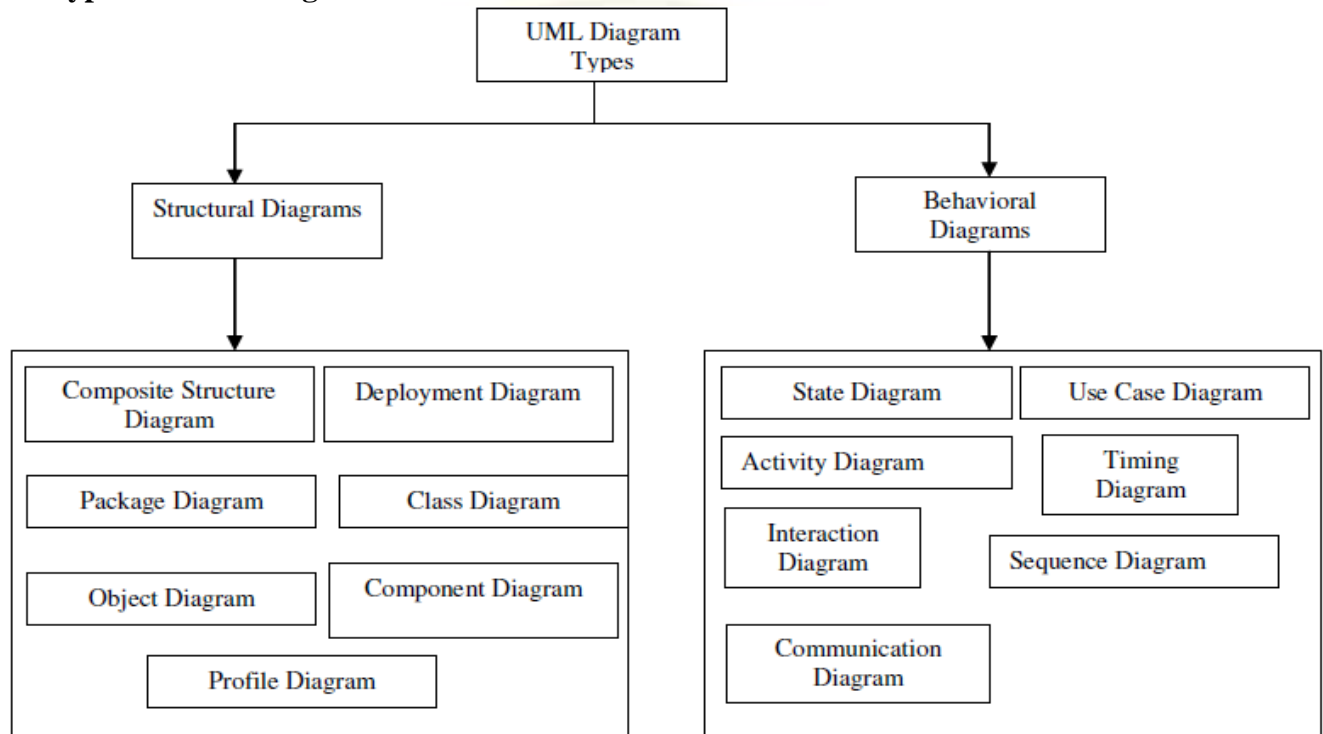
- UML stands for “Unified Modeling Language”
- It is a industry-standard graphical language for specifying, visualizing, constructing, and documenting the artifacts of software systems
- The UML uses mostly graphical notations to express the OO analysis and design of software projects.
- Simplifies the complex process of software design

#### **3.1 Why UML for Modelling**

- Use graphical notation to communicate more clearly than natural language (imprecise) and code (too detailed).
- Help acquire an overall view of a system.
- UML is *not* dependent on any one language or technology.
- UML moves us from fragmentation to standardization.

**List various UML diagrams and explain the purpose of each diagram. (May/June 2014, May/June 2016, April/May 2017)**

### 3.2 Types of UML Diagrams



#### Structure Diagrams

- Structure diagrams emphasize on the things that must be present in the system being modeled.
- Since structure diagrams represent the structure, they are used extensively in documenting the software architecture of software systems.

#### Behavioural Diagrams

- Behavior diagrams emphasize on what must happen in the system being modeled.
- Since behavior diagrams illustrate the behavior of a system, they are used extensively to describe the functionality of software systems.

#### 4. USE CASE DIAGRAM

**For any scenario draw the use case diagram in detail and explain. (May/June 2015, April/May2011, May/June 2012,May/June 2014)**

**Define use case diagrams.(nov/dec 2011, may/june 2012)**

- A use case diagram describes how a system interacts with outside actors.
- It is a graphical representation of the interaction among the elements and system.
- Each use case representation a piece of functionality that a system provides to its user.
- Use case identifies the functionality of a system.

A use case diagram contains four components.

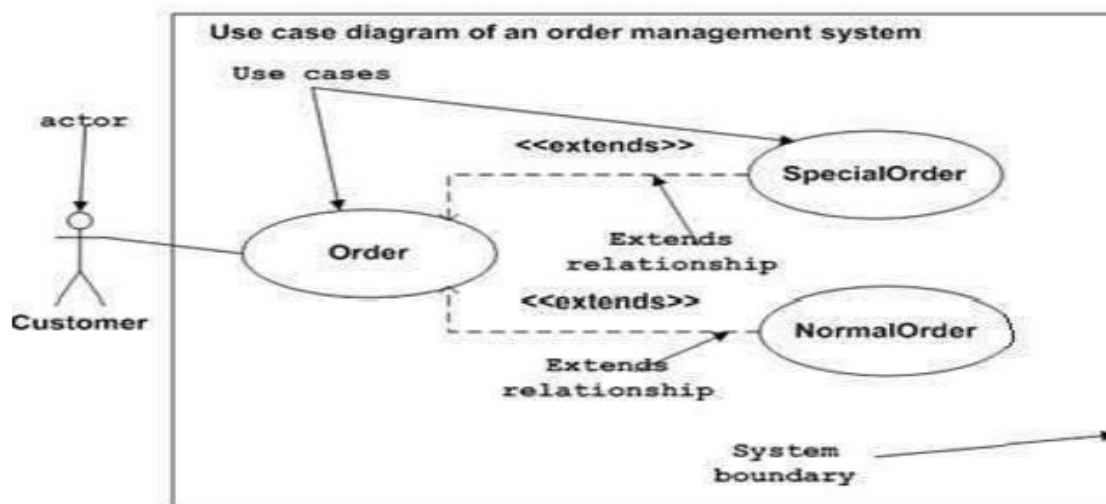
- The **boundary**, which defines the system of interest in relation to the world around it.
- The **actors**, usually individuals involved with the system defined according to their roles.
- The **use cases**, which the specific roles are played by the actors within and around the system.
- The **relationships between** and among the actors and the use cases.

**Purpose:**

- The main purpose of the use case diagram is to capture the dynamic aspect of a system.
- Use case diagram shows, what software is suppose to do from user point of view.
- It describes the behavior of system from user's point.
- It provides functional description of system and its major processes.
- Use case diagram defines the scope of the system you are building.

**When to Use: Use Cases Diagrams**

- Use cases are used in almost every project.
- They are helpful in exposing requirements and planning the project.
- During the initial stage of a project most use cases should be defined.



**Figure: Sample Use Case diagram**

## 5. CLASS DIAGRAM

**For any scenario draw the class diagram in detail and explain. (May/June 2012, 2015, Nov/Dec 2011)**

### Introduction

- The class diagram is a static diagram.
- A class model captures the static structure of a system by characterizing the objects in the system, the relationship between the objects, and the attributes and operations for each class of objects.
- The class diagram can be mapped directly with object oriented languages.
- Class diagram provide a graphical notation for modeling classes and their relationship.

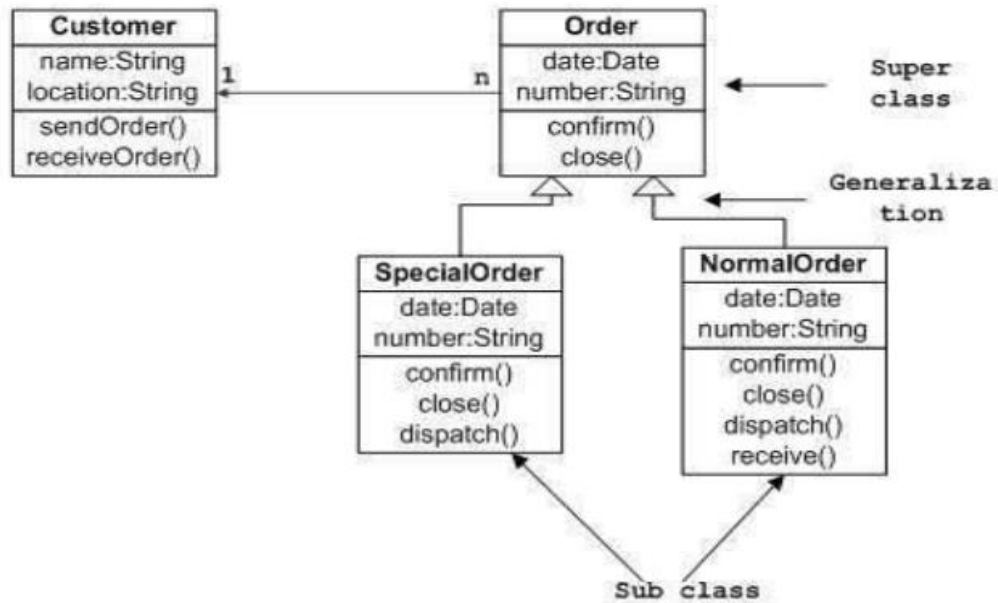
### Purpose

- Analysis and design of the static view of an application.
- Describe responsibilities of a system.
- Base for component and deployment diagrams.

### When to use: Class Diagram

- Useful for Forward and Reverse engineering.
- Class diagrams are useful both for abstract modeling and for designing actual programs.
- Developer uses class diagram for implementation decision.

### Sample Class Diagram



## 6. STATE CHART DIAGRAM

**For any scenario draw the state chart/ state machine diagram in detail and explain.(May/June 2015,2014,Nov/Dec2011)**

### Introduction

- A **state diagram** is a graph in which nodes correspond to states and directed arcs correspond to transitions labeled with event names.
- A state diagram combines states and events in the form of a network to model all possible object states during its life cycle, helping to visualize how an object responds to different stimuli.
  - o A state diagram is a graph whose nodes are states and whose directed arcs are transitions between states.
  - o A state diagram specifies the state sequence caused by event sequence.

### Purpose

- The state model describes those aspects of objects concerned with time and the sequencing of operations events that mark changes, states that define the context for events, and the organization of events and states.
  - They are used to give an abstract description of the behavior of a system.
  - It provides direction and guidance to the individual counties within the states.
  - It specifies the possible states, what transitions are allowed between states.
  - It describes the common behavior for the objects in a class and each object changes its behavior from one state to another.

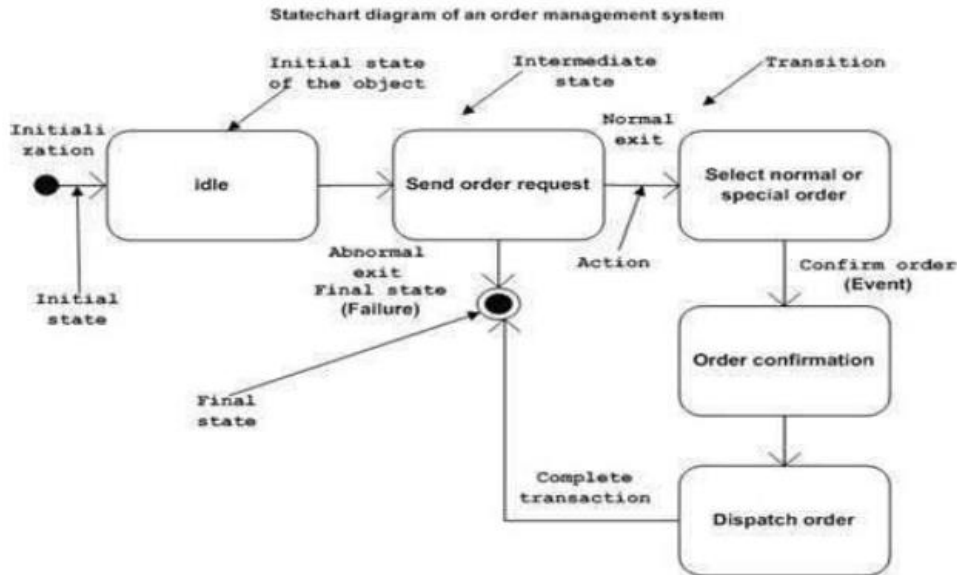
### When to use: State Diagram

- o They are perfectly useful to model behavior in real time system.



- o Each state represents a named condition during the life of an object during which it satisfies some condition or waits for some event.
- o It determines how objects of that class react to events.
- o For each object state, it determines what actions the object will perform when it receives an event.

### Sample State Diagram



## 7. INTERACTION DIAGRAM

For any scenario draw the interaction diagram in detail and explain. (May/June 2011,2015,Nov/Dec 2012)

This interactive behaviour is represented in UML by two diagrams known as

- Sequence Diagram
- Collaboration Diagram

### 7.1 SEQUENCE DIAGRAM (INTERACTION DIAGRAM)

#### Introduction

- Sequence diagrams model the dynamic aspects of a software system.
- The emphasis is on the “sequence” of messages rather than relationship between objects.
- A sequence diagram maps the flow of logic or flow of control within a usage scenario into a visual diagram enabling the software architect to both document and validate the logic during the analysis and design stages.
- Sequence diagrams provide more detail and show the message exchanged among a set of objects over time.

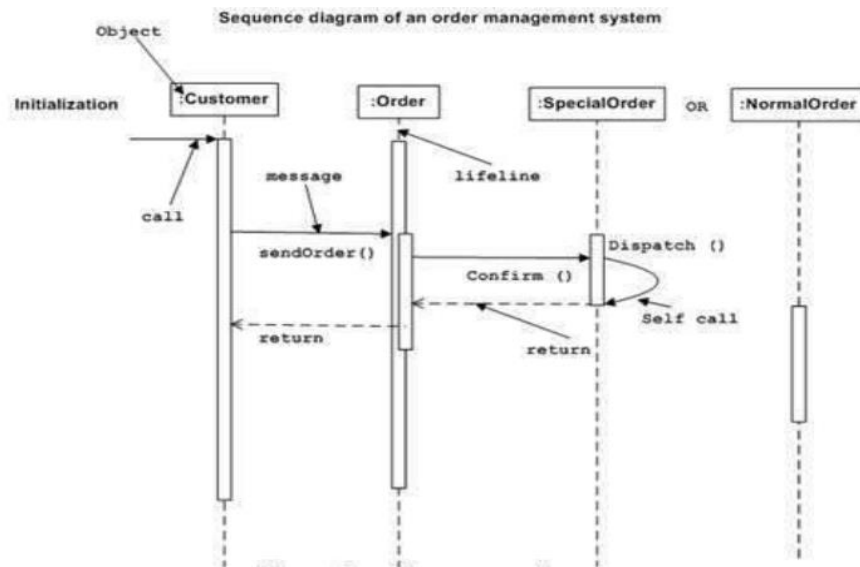
#### Purpose

- The main purpose of this diagram is to represent how different business objects interact.

- A sequence diagram shows object interactions arranged in time sequence.
- It depicts the objects and classes involved in the scenario and the sequence of messages
- Exchanged between the objects needed to carry out the functionality of the scenario.

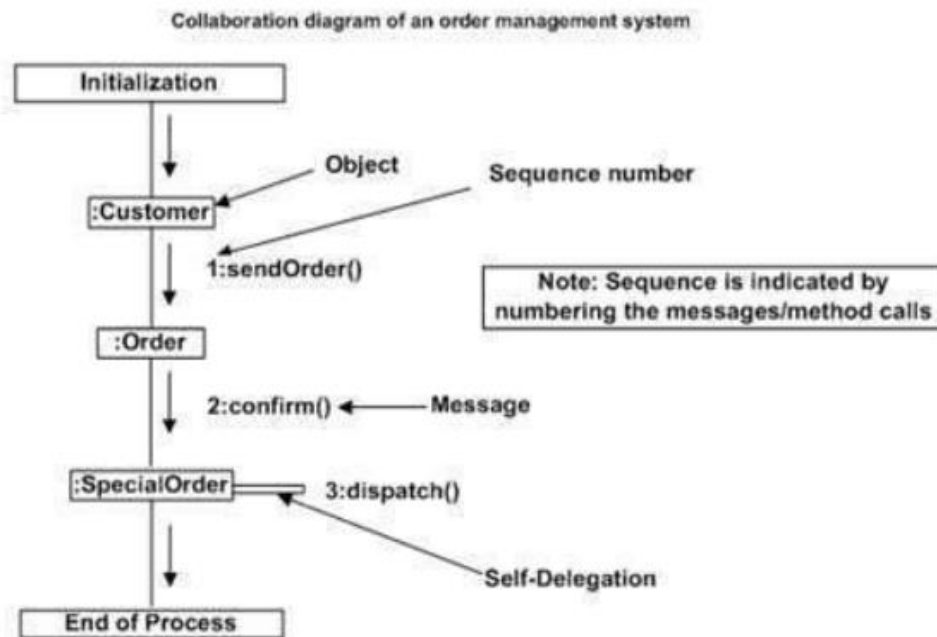
#### When to use: Sequence Diagram

- Sequence diagram can be a helpful modeling tool when the dynamic behavior of objects needs to be observed in a particular use case or when there is a need for visualizing the “big picture of message flow”.
- A company’s technical staff could utilize sequence diagrams in order to document the behavior of a future system.



## 7.2 COLLABORATION DIAGRAM (INTERACTION DIAGRAM)

- The second interaction diagram is collaboration diagram. It shows the object organization as shown below. 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. We have taken the same order management system to describe the collaboration 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.



## 8. ACTIVITY DIAGRAM

**For any scenario draw the activity diagram in detail and explain.(May/June 2011, 2015, Nov/Dec 2011)**

### Introduction

- An activity diagram is a type of flow chart with additional support for parallel behavior.
- This diagram explains overall flow of control.
- Activity diagram is another important diagram in UML to describe dynamic aspects of the system.
- Activity diagram is basically a flow chart to represent the flow from one activity to another activity

### Purpose

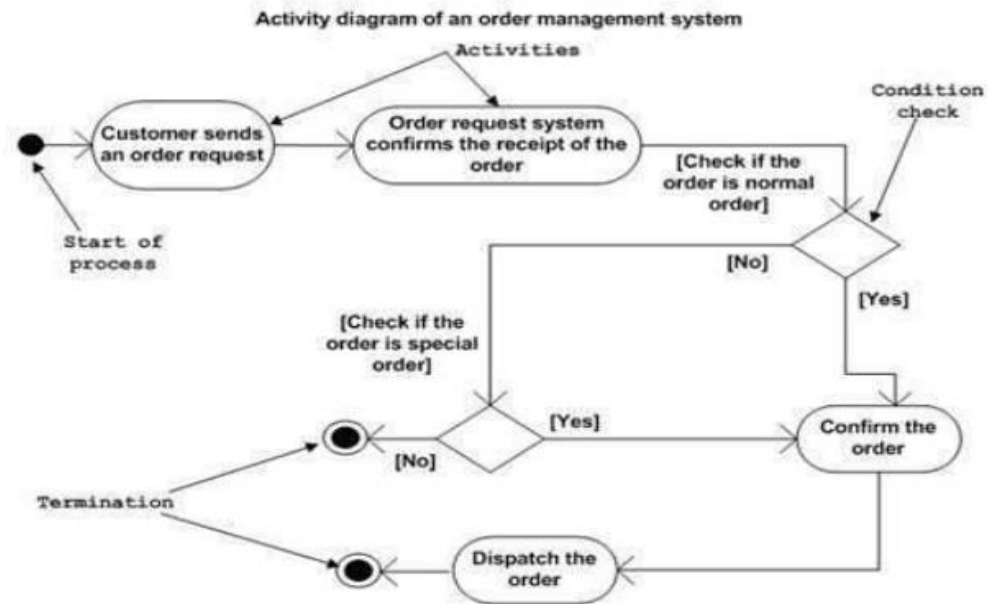
- Contrary to use case diagrams, in activity diagrams it is obvious whether actors can perform
- Business use cases together or independently from one another.
- Activity diagrams allow you to think functionally.

### When to use: Activity Diagrams

- Activity diagrams are most useful when modeling the parallel behavior of a multithreaded system or when documenting the logic of a business process.
- Because it is possible to explicitly describe parallel events, the activity diagram is well suited for the illustration of business processes, since business processes rarely occur in a linear manner and often exhibit parallelisms.

- This diagram is useful to investigate business requirements at a later stage.

### Sample Activity Diagram



## 9. DEPLOYMENT DIAGRAM

**For any scenario draw the implementation diagram ( component and deployment) in detail and explain. (May/June 2011, 2012, 2014, 2015)**

Deployment diagrams are used to visualize the topology of the physical components of a system where the software components are deployed. So deployment diagrams are used to describe the static deployment view of a system. Deployment diagrams consist of nodes and their relationships.

### Purpose:

The purpose of deployment diagrams can be described as:

- Visualize hardware topology of a system.
- Describe the hardware components used to deploy software components.
- Describe runtime processing nodes.

### How to draw Deployment Diagram?

Deployment diagram represents the deployment view of a system. It is related to the component diagram. Because the components are deployed using the deployment diagrams. A deployment diagram consists of nodes. Nodes are nothing but physical hardwares used to deploy the application.

Deployment diagrams are useful for system engineers. An efficient deployment diagram is very important because it controls the following parameters

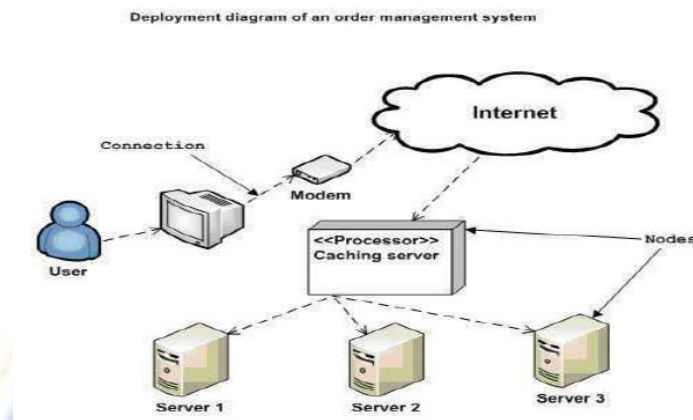
- Performance
- Scalability
- Maintainability

- Portability

So before drawing a deployment diagram the following artifacts should be identified:

- Nodes
- Relationships among nodes

The following deployment diagram is a sample to give an idea of the deployment view of order management system



### Where to use Deployment Diagrams?

Deployment diagrams are mainly used by system engineers. These diagrams are used to describe the physical components (hardwares), their distribution and association.

So the usage of deployment diagrams can be described as follows:

- To model the hardware topology of a system.
- To model embedded system.
- To model hardware details for a client/server system.
- To model hardware details of a distributed application.
- Forward and reverse engineering.

## 10. COMPONENT DIAGRAM

Component diagrams are different in terms of nature and behaviour. Component diagrams are used to model physical aspects of a system.

### Purpose:

Component diagram is a special kind of diagram in UML. The purpose is also different from all other diagrams discussed so far. It does not describe the functionality of the system but it describes the components used to make those functionalities.

A single component diagram cannot represent the entire system but a collection of diagrams are used to represent the whole.

So the purpose of the component diagram can be summarized as:

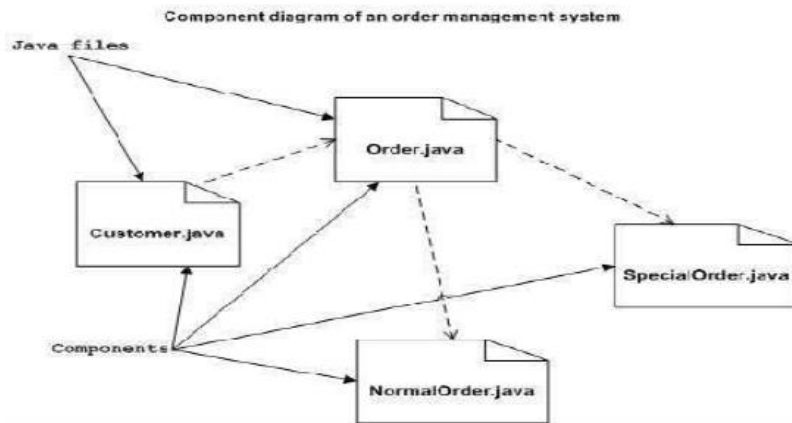
- Visualize the components of a system.
- Construct executables by using forward and reverse engineering.
- Describe the organization and relationships of the components.

### How to draw Component Diagram?



Component diagrams are used to describe the physical artifacts of a system. This artifact includes files, executables, libraries etc. So the purpose of this diagram is different, Component diagrams are used during the implementation phase of an application.

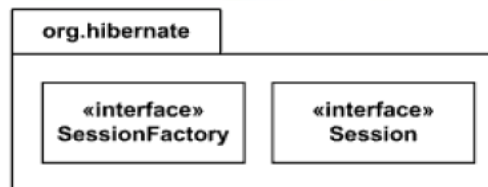
So the following component diagram has been drawn considering all the points mentioned above:



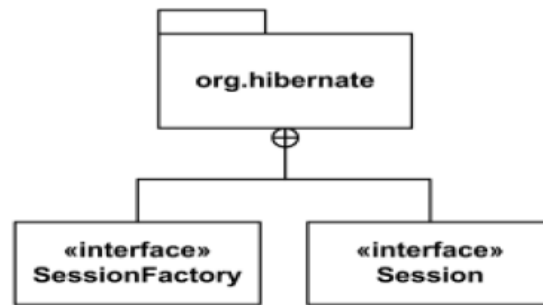
## 11. PACKAGE DIAGRAM

**For any scenario draw the Package diagram in detail and explain. (May/June 2014,2015)**

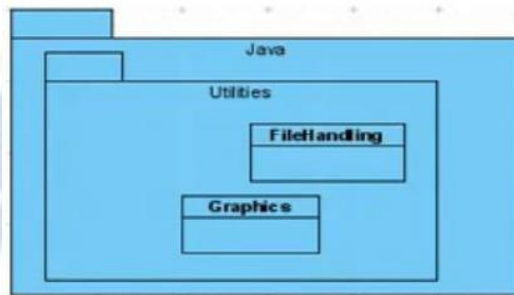
- A package is used to group elements, and provides a namespace for the grouped elements. A package is a namespace for its members, and may contain other packages.
- Owned members of a package should all be package elements.
- Package can also be merge with other package, thus provide the hierarchical organization of the package.
- The elements that can be referred to within a package using non-qualified names are: Owned Element, Imported Element, and elements enclosing namespaces.
- Owned and imported elements may have a visibility that determines whether they are available outside the package.
- Package Member are not shown inside the package.
- Package org.hibernate contains two members Session Factory and Session inside the package as shown below.



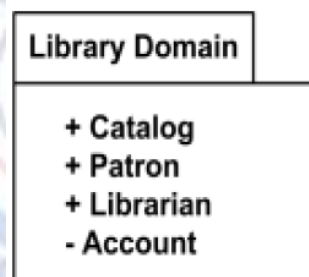
- Members of the package may be shown outside of the package by branching lines.
- Package org.hibernate contains interfaces Session and Session Factory.



- Packages are useful for simplify this kind of diagrams
- Nested packages.
- Qualifier for Graphics class is Java::Utilities::Graphics



- Visibility of Owned and Import element.
- "+" for public and "-" for private or helper class.
- All elements of Library Domain package are public except for Account.



- Package able element is a named element that may be owned directly by a package.
- Owned member of the package should all be package able elements.
- If a package is removed from the model, so are all the elements owned by the package. Package by itself is package able element, so any package could be also a member of the other packages.

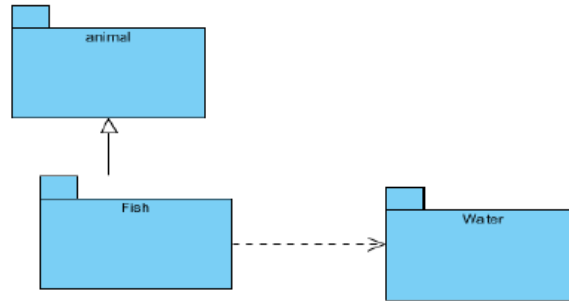
### Relationships in Package Diagram

- Dependency
- Generalization
- Refinement

### Dependency

- One Package depends on another package.

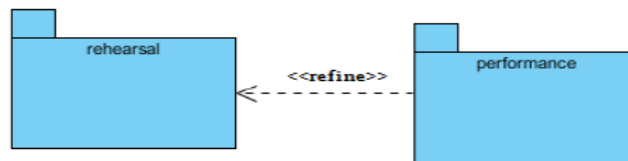
### Generalization



Fish is a kind of Animal

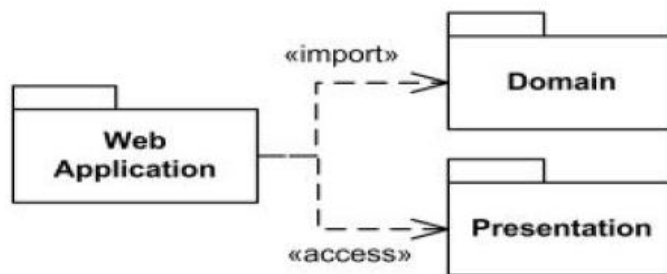
### Refinement

- Refinement shows different kind of relationship between packages.
- One Package refines another package, if it contains same elements but offers more details about those elements.



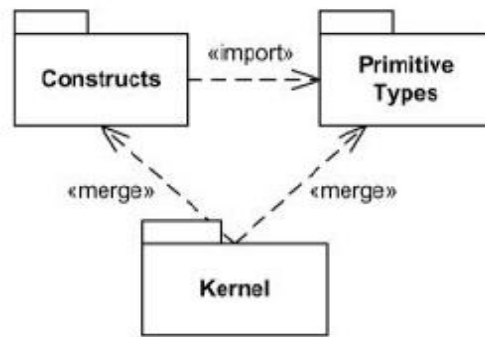
### Package Import

- Private import of Presentation package and public import of Domain package.



### Package Merge

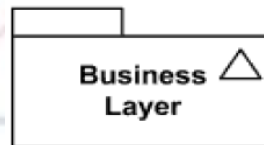
- A package merge is a directed relationship between two packages.
- It indicates that content of one package is extended by the contents of another package.
- Package merge used when elements defined in different packages have the same name and are intended to represent the same concept.
- Package merge is shown using a dashed line with an open arrowhead pointing from the receiving package to the merged package.



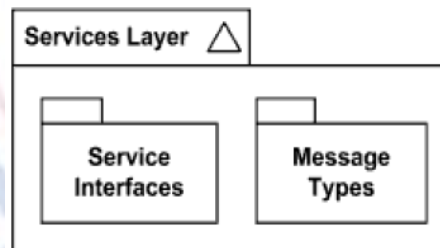
### Model

- Model is a package which captures a view of a system.
- View of the system defined by its purpose and abstraction level.
- Model is notated using the ordinary package symbol (a folder icon) with a small triangle in the upper right corner of the large rectangle.

### *Business layer model:*

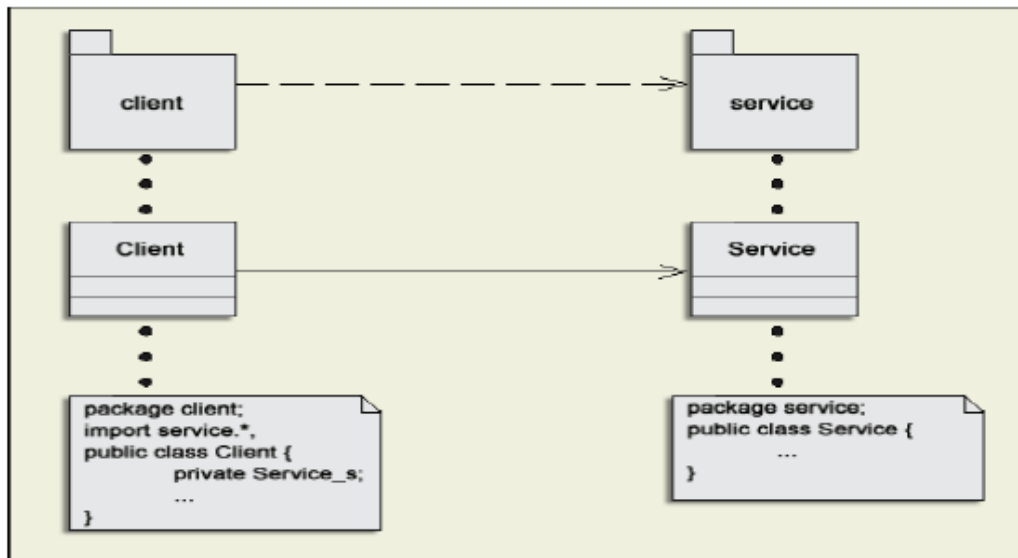


- If contents of the model are shown within the large rectangle, the triangle may be drawn to the right of the model name in the tab.
- Service Layer model contains service interfaces and message types.



### Package Relationship

- A relationship between two packages is called a package dependency.
- Dependencies are not transitive.
- The dependency relationship between packages is consistent with the associative relationship between classes.



### When to Use Package Diagram

- A large complex project can have hundreds of classes. Without some way to organize those classes, it becomes impossible to make sense of them all.
- Packages create a structure for your classes or other UML elements by grouping related elements.

### Use of Package Diagram

- When you want to show a high-level view of the system.
- To keep track of dependencies.
- With a large system to show its major elements and how they relate to one another.
- To divide a complex system into modules.
- Package diagrams can use packages that represent the different layers of a software system to illustrate the layered architecture of a software system.



## CS6502- OBJECT ORIENTED ANALYSIS AND DESIGN

### UNIT – II DESIGN PATTERNS

GRASP: Designing objects with responsibilities – Creator – Information expert – Low Coupling – High Cohesion – Controller - Design Patterns – creational - factory method - structural – Bridge – Adapter - behavioral – Strategy – observer.

---

#### 1. GRASP INTRODUCTION

##### What is GRASP? (Nov/Dec 2011, May/June 2015, 2016)

##### 1. GRASP:

**GRASP** stands for **General Responsibility Assignment Software Patterns**, consists of guidelines for assigning responsibility to classes objects in object-oriented design.

##### 1.1 GRASP as a Methodical Approach to Learning Basic Object Design

- It is possible to communicate the detailed principles and reasoning required to grasp basic object design, and to learn to apply these in a methodical approach that removes the magic and vagueness.
- The GRASP patterns are a learning aid to help one understand essential object design, and apply design reasoning in a methodical, rational, explainable way.
- This approach to understanding and using design principles is based on *patterns of assigning responsibilities*.

##### What is responsibilities? What are the two responsibilities?

##### 1.2 Responsibilities and Methods

- Responsibility is defined as a contract or obligation of a type or class and is related to behavior.
- There are 2 types of responsibilities
  1. **Knowing** - responsibilities of an object include
    - a. Knowing about private encapsulated data-member data
    - b. Knowing about related objects
    - c. Knowing about things it can derive or calculate
  2. **Doing** - responsibility of an object include
    - a. Doing something itself-assign, calculate and create
    - b. Initiating action in other objects
    - c. Controlling and coordinating activities in other objects

#### 2. Patterns

**Explain in detail on Grasp Pattern. (April/ May 2011, May/June 2015,2016)**

**Define pattern.**

Experienced object-oriented developers (and other software developers) build up a repertoire of both general principles and idiomatic solutions that guide them in the creation of software. These principles and idioms, if codified in a structured format describing the problem and solution, and given a name, may be called **patterns**.

In object technology, a **pattern** is a named description of a problem and solution that can be applied to new contexts; ideally, it provides advice in how to apply it in varying circumstances, and considers the forces and trade-offs.

**2.1 Patterns Have Names**

All patterns ideally have suggestive names. Naming a pattern, technique, or principle has the following advantages:

- It supports chunking and incorporating that concept into our understanding and memory.
- It facilitates communication.

Naming a complex idea such as a pattern is an example of the power of abstraction-reducing a complex form to a simple one by eliminating detail.

**2.2 GRASP: Patterns of General Principles in Assigning Responsibilities**

To summarize the preceding introduction:

- The skillful assignment of responsibilities is extremely important in object design.
- Determining the assignment of responsibilities often occurs during the creation of interaction diagrams, and certainly during programming.

**2.3 How to Apply the GRASP Patterns?**

The following sections present the first five GRASP patterns:

- Information Expert
- Creator
- High Cohesion
- Low Coupling
- Controller

**3. Information Expert**

---

**Explain in detail about information expert.(April/May 2017)**

**Problem:** Any real world application has hundreds of classes and thousand of actions. How do I start assigning the responsibility?

**Solution:** Assign a responsibility to Information Expert, the class that has information necessary to fulfill the responsibility.”

Generally, if domain model is ready then it is pretty straight forward to assign the responsibilities and arrive at design model.

**Approach**

**Step I:** State the responsibility clearly.

**Step II:** Search for the classes who have the information needed to fulfil the responsibility.

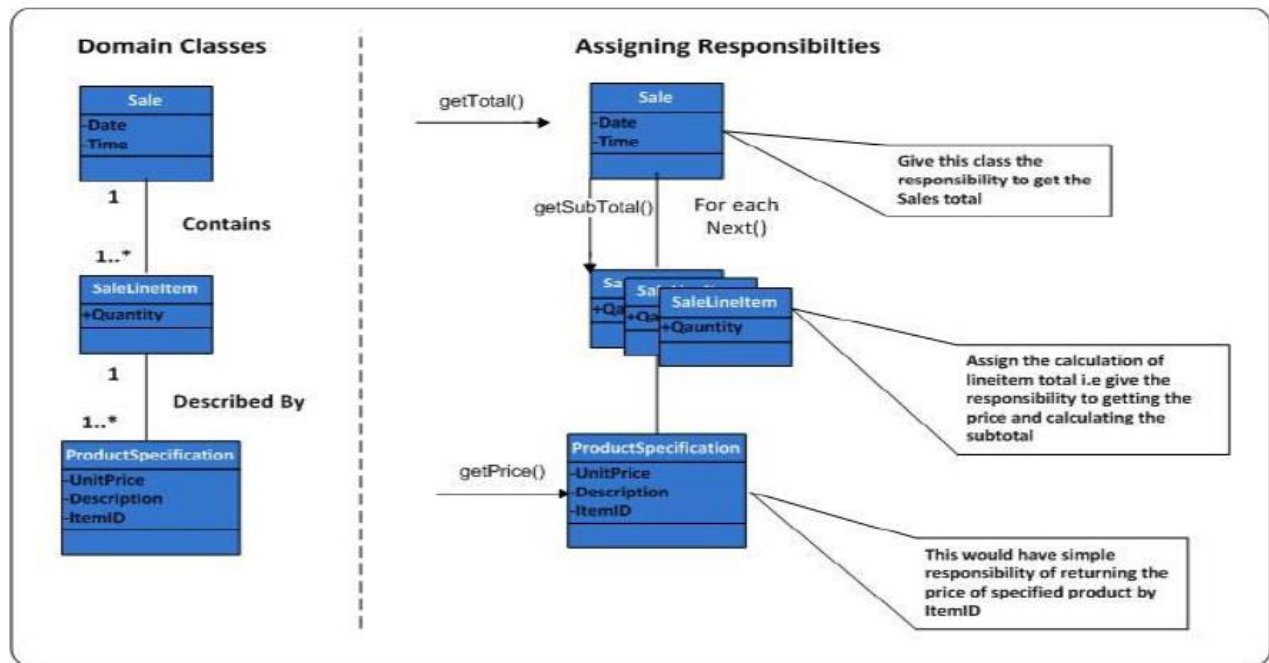
**Step III:** Assign the responsibility

**Description**

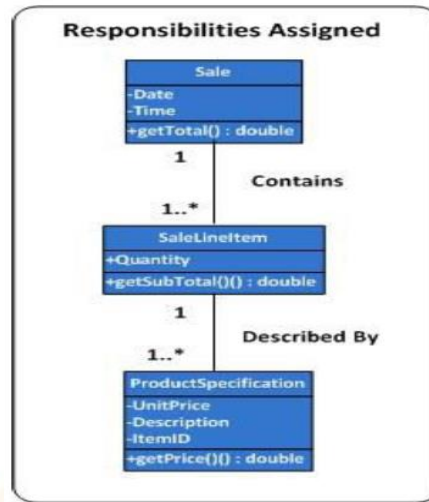
Let’s take an example of POS (Point Of Sale) systems. This is what we come across malls where there are many POS counters. In the domain model, following classes are identified. These classes are shown in the left part of diagram (Fig. No.1). Let’s think about some responsibility.

**Who should have the responsibility of calculating the Sales Total?**

Sales Total =Sum of (Unit price X quantity of sale line item) E.g. a person is buying 2 line items, toothpaste (2 no’s) and soap (2 no’s) are being bought Sales total = (Price of toothpaste)\* (quantity of toothpaste) + (Price of soap)\* (quantity of soap)



- It is demonstrated in right portion of above diagram (Fig. No.1).
- Following diagram depicts the assignment of responsibilities and corresponding methods.



#### Benefits:

- Encapsulation is maintained as objects use their own data to fulfil the task.
- Support low coupling
- Behaviour (responsibility) is distributed across the classes that have the required information thus encouraging the highly cohesive lightweight classes

#### Liabilities /Contradictions:

- Sometimes while assigning responsibilities, lower level responsibilities are uncovered
- Not all the times the application of this pattern is desirable.

#### **4. Creator**

##### Describe the concept of creator. (April/May 2012, 2017)

**Problem:** Who creates the new instance of some class?

##### **Solution:**

Assign class A the responsibility to create an instance of class B if....

- A aggregates (whole-part relationship) B objects
- A contains B objects
- A records instances of B objects
- A closely uses B objects
- A has initializing data that is needed while creating B objects (thus A is an expert with respect to creating B)

##### **Approach:**

**Step I:** Closely look at domain/ design model and ask: who should be creating these classes?

**Step II:** Search for the classes who create, aggregate etc.

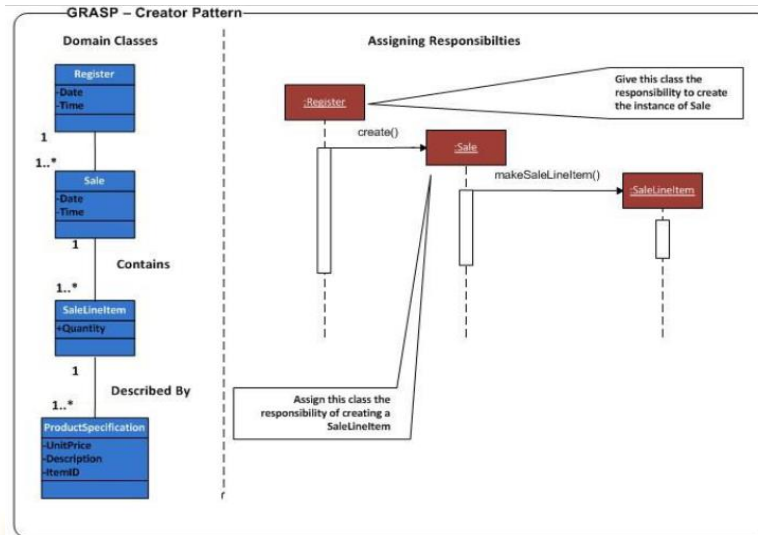
**Step III:** Assign the responsibility of creation

##### **Description**



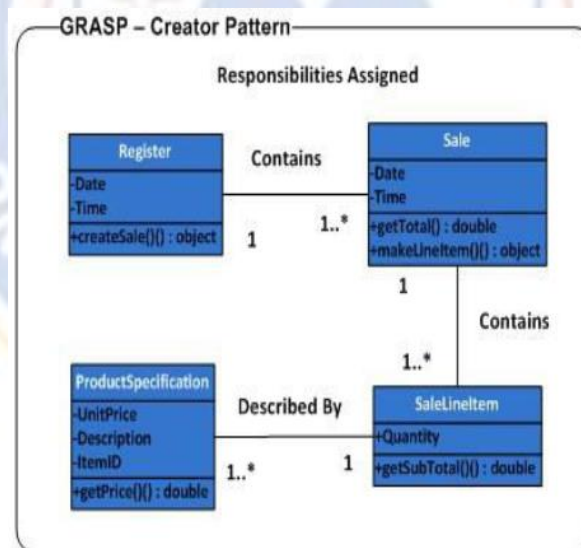
Let's extend an example of POS (Point Of Sale) systems explained in previous part. This is what we come across malls where there are many POS counters. In the domain model, following classes are identified. These classes are shown in the left part of diagram (Fig. No.1).

**Fig. No. 1**



Following diagram depicts the assignment of responsibilities and corresponding methods.

**Fig No. 2**



**Benefits:**

- Support low coupling
- The responsibility is assigned to a class which fulfils the criteria thus separating the logic for creation and actual execution.

**Liabilities /Contradictions:**

- Sometimes creation may require significant complexity. Many a times there would be tight coupling among the creator and create class. Also the task of destroying the objects is to



be thought of. Recycling of such instances (pooling) becomes important for the performance reasons.

- In many scenarios the creations is conditional and it may be from family of classes making it very complex.
- For creation of family of classes or to tackle the creation and recycling, other patterns are available which are factory, abstract factory etc.

## **5. Controller**

### **Explain about controller. (May/June 2012, 2016, 2017, Nov/Dec 2015)**

**Problem:** Who should be responsible for handling a system event?

**Solution:** Assign the responsibility for handling a system event message to a class representing one of the following

- A class that represents the overall system, device, or sub-system (facade controller)
- A class that represents a use case within which the system event occurs.
- Represents the overall business (facade controller)
- Represents something in real world that is active (role controller)

These classes often don't do the work themselves, but delegate it to others and in other terms it coordinates or controls the activity.

The decision to create the specific controller i.e. system controllers vs. use case controllers are often driven and influenced by the dynamics of high cohesion vs. low coupling scenario.

#### **Approach:**

**Step I:** Closely look at domain/ design model and when you have family of classes doing work/job pertaining to some specific function and would need some facilitator e.g. if one has requirement to connect to multiple databases and has the DAL, a Communication Manager class can be introduced to control and manage DAL and connection to different database

**Step II:** Add a new class to take the responsibility of controller or manager and all client code would access the functionality through this class

#### **Description**

Let's extend an example of POS (Point Of Sale) systems explained in previous part. In the domain model, following classes are identified. These classes are shown in the left part of diagram (Fig. No.1). There are many system operations such as closing a sale, make payment etc which are related to business function "Sales".

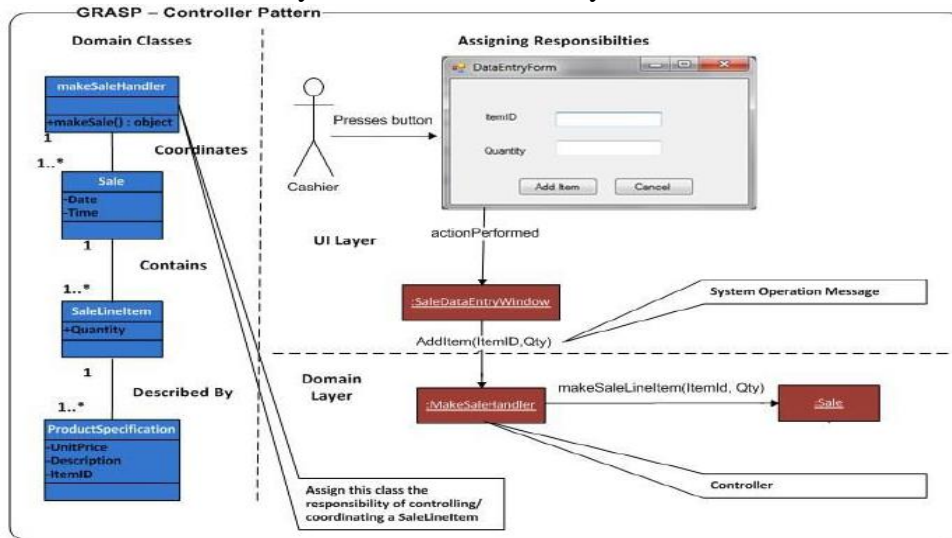
Controller pattern gives following choices

- represents overall system
- represents the overall business
- MakeSaleHandler- an artificial handler of all system operations

Following diagram depicts the Controller pattern. Compared to Creator pattern, it has additional class called makeSaleHandler which controls and coordinates the sale.

**Cashier** - represents some role in real-world that is active in the task

**UI Form**- shows the UI layer for POS software system



**Bloated Controller:** There are the controllers which handle too many system events leading to low cohesion. This would happen when only one class is receiving all system events.

These perform many of the tasks for handling events without delegating the work and also has significant amount of information which should have distributed to other project.

Sometimes it duplicates information found somewhere else. Such controller better be avoided and they can be avoided by addition of few more controllers.

Design the controller in such a way that it primarily delegates the fulfillment of each system operation to other objects.

**Related Patterns:** This patterns needs to be studied from the angle of understanding and applying principle. The patterns related to this are “Low Coupling” and “High Cohesion”.

**Benefits:**

- Supports Low Coupling
- Promotes Understandability, maintainability

**Liabilities /Contradictions:**

- Sometimes grouping of responsibilities or code into one class or component to simplify maintenance occurs when controllers are designed. This should be avoided as it would become bloated controllers as discussed above.

## 6. Low Coupling

Explain about low coupling. (May/June 2012,2016,2017, Nov/Dec 2015)

**Objective**

- Coupling refers to connectedness or linking. It is a measure of how strongly one class is connected to or has knowledge (information or knowhow) of or relies/depends upon the other classes.
- With experience, programmers would agree on something i.e. making changes is always been the hardest task and given a chance all would like to work upon systems from ground up.
- Designing for low connectedness is an important activity which can help to have a system in such a way that the changes in one element (sub-system, system, class, etc) will limit changes to other elements.

**Problem:** How to reduce impact of changes and encourage reuse?

**Solution:** Assign a responsibility so that the coupling (linking classes) remains low

**Approach:**

**Step I:** Closely look for classes with many associations to other classes.

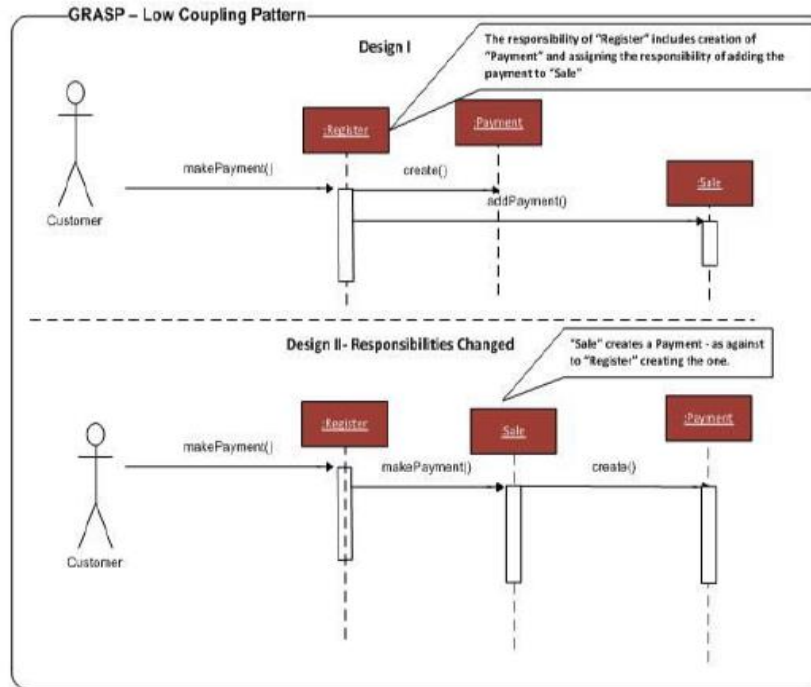
**Step II:** Look for the methods that rely on a lot of other methods (or methods in other classes i.e. dependencies)

**Step III:** Rework the design so as assign responsibilities to different classes in such a way that they have lesser association and dependency.

**Description**

Let's extend an example of POS (Point Of Sale) systems carried in series of articles. In the domain model, following classes like Register, Payment and Sale are identified. These classes are shown in the left part of diagram (Fig. No.3). There are many system operations such as closing a sale, make payment etc which are related to business function "Sales".

**Fig No. 3**



The common forms of coupling for A to B include

- A has a data member which refers to B instance
- A has a method which B is an argument
- A is direct or indirect subclass of B
- B is an interface and A implements B

#### Related Patterns:

The pattern related to is "High Cohesion".

#### Benefits

- Maintainability - Little or not affected by changes in other components
- Understandability- simple to understand in the isolation
- Reusability- convenient to reuse and easier to grab hold of classes

#### Liabilities / Contradictions:

- Coupling to stable elements (classes in library or well tested classes) doesn't account to be a problem.

### 7. High Cohesion



**Explain about high cohesion.(may/june 2012,2016,2017, nov/dec 2015)**

**Objective**

- The dictionary meaning of the word “cohesion” is “the state of cohering or sticking together”. In botany world “the process of some parts (which are generally separate) of plant growing together” and in physics world “it’s the intermolecular force that holds together the molecules in solid or liquid”.
- Cohesion can be said to a measure of “relatedness” or “how related all the attributes and behaviour in a class are”. High cohesion refers to how strongly related the responsibilities are. A class with low cohesion i.e. carrying out the unrelated responsibilities in turn does so many unrelated things or end up doing too much of a work.

**Problem:** How to keep classes focused and manageable?

**Solution:** Assign responsibility so that cohesion remains high

**Approach:**

**Step I:** Closely look for classes with probability of many responsibilities i.e. classes with too-few or disconnected methods.

**Step II:** Look for the methods which do a lot (look for method names having words like “AND”, “OR”)

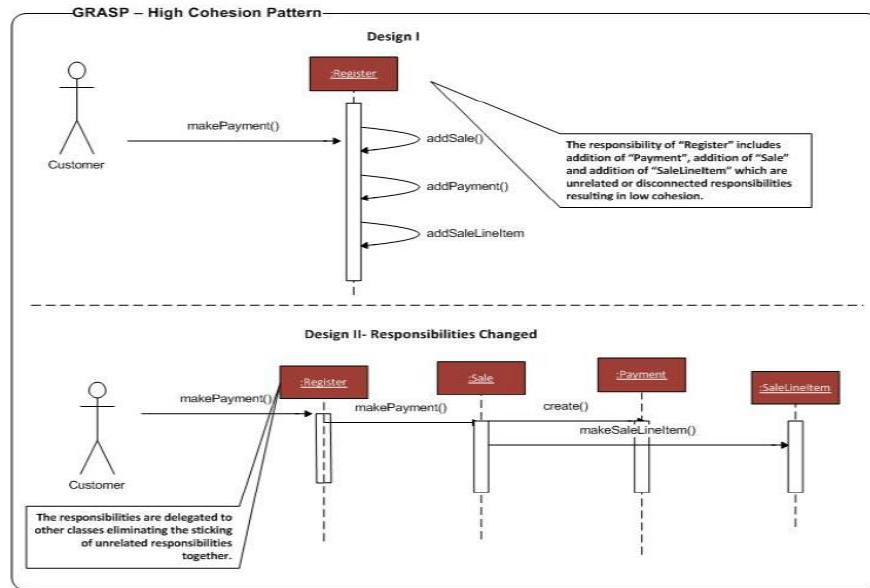
**Step III:** Assign a responsibility in such a way that all the responsibilities in a class are related

**Description**

Let’s extend an example of POS (Point Of Sale) systems carried in series of articles. In the domain model, following classes like Register, Payment and Sale are identified. In the upper part of diagram (Fig.No.4) a class “Register” is displayed carrying out many unrelated responsibilities. There are many system operations such as closing a sale, make payment etc which are related to business function “Sales”.

**Fig No. 4**





**Related Patterns:** This pattern is related rather complementary to “Low Coupling” and little Contradictory to information expert.

**Benefits:**

- Understandability :- Clarity and ease of understanding design is increased
- Maintainability :- Maintenance and enhancements are simplified
- Low coupling is often supported
- Reusability :- Small size classes with highly related functionality increases reuse

**Liabilities / Contradictions:**

- In scenarios of server objects, it would be desirable to have less cohesive server objects due to performance needs. Remote objects and remote communication are the examples where performance matters than anything else. In this case, an interface provided for many operations would be recommended than having highly cohesive but different interfaces/ components.

- While grouping the responsibilities or code into one class or component motivated by maintainability from view of one person is not desirable as it would end up in maintainability for one person but not for all. This happens when the responsibilities could be related to class but not to each other. This is a good trap for any developer who can never imagine there would be somebody else working on it and he/she has preferred to keep things at one place for better traceability.

**8. Design Patterns**

**Explain the design patterns and the principles used in it.(nov/dec 2011, may/june 2015,2016)**

A design patterns are **well-proved solution** for solving the specific problem/task.

**Problem Given:**

Suppose you want to create a class for which only a single instance (or object) should be created and that single object can be used by all other classes.

**Solution:**

**Singleton design pattern** is the best solution of above specific problem. So, every design pattern has **some specification or set of rules** for solving the problems. What are those specifications, you will see later in the types of design patterns.

**Advantage of design pattern:**

1. They are reusable in multiple projects.
2. They provide the solutions that help to define the system architecture.
3. They capture the software engineering experiences.
4. They provide transparency to the design of an application.
5. They are well-proved and testified solutions since they have been built upon the knowledge and experience of expert software developers.
6. Design patterns don't guarantee an absolute solution to a problem. They provide clarity to the system architecture and the possibility of building a better system.

**When should we use the design patterns?**

We must use the design patterns **during the analysis and requirement phase of SDLC** (Software Development Life Cycle).

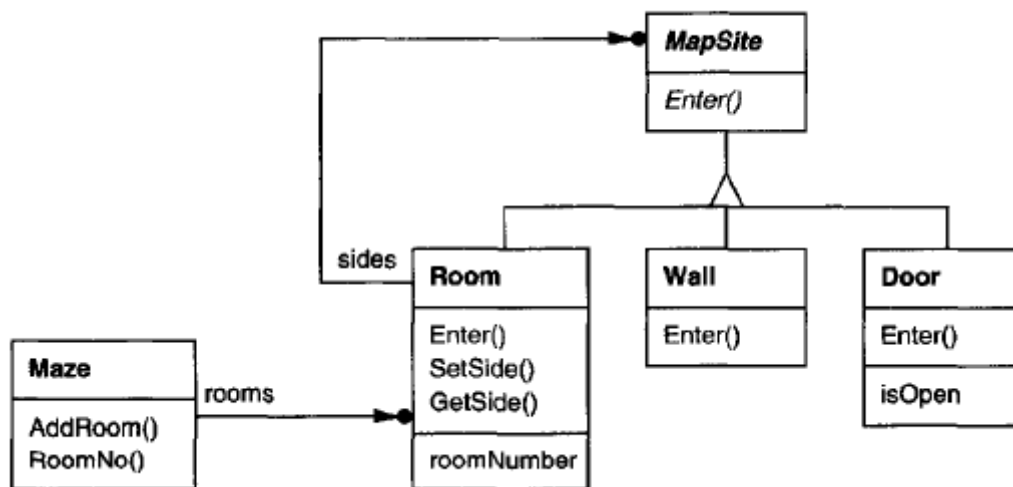
Design patterns ease the analysis and requirement phase of SDLC by providing information based on prior hands-on experiences.

**9. Creational Pattern**

**Explain in detail on creational pattern.(Nov/Dec 2015)**

- Creational design patterns abstract the instantiation process. They help make a system independent of how its objects are created, composed, and represented.
- A class creational pattern uses inheritance to vary the class that's instantiated, whereas an object creational pattern will delegate instantiation to another object.
- Creational patterns become important as systems evolve to depend more on object composition than class inheritance.
- As that happens, emphasis shifts away from hard coding a fixed set of behaviors toward defining a smaller set of fundamental behaviors that can be composed into any number of more complex ones.
- Thus creating objects with particular behaviors requires more than simply instantiating a class.
- There are two recurring themes in these patterns.

- First, they all encapsulate knowledge about which concrete classes the system uses.
- Second, they hide how instances of these classes are created and put together. All the system at large knows about the objects is their interfaces as defined by abstract classes.
- Consequently, the creational patterns give you a lot of flexibility in *what* gets created, *who* creates it, *how* it gets created, and *when*. They let you configure a system with "product" objects that vary widely in structure and functionality.
- Configuration can be static (that is, specified at compile-time) or dynamic (at run-time).
- We'll also use a common example— building a maze for a computer game—to illustrate their implementations. The maze and the game will vary slightly from pattern to pattern. Sometimes the game will be simply to find your way out of a maze; in that case the player will probably only have a local view of the maze.
- Sometimes mazes contain problems to solve and dangers to explored. We'll ignore many details of what can be in a maze and whether a maze game has a single or multiple players. Instead, we'll just focus on how mazes get created. We define a maze as a set of rooms.
- A room knows its neighbors; possible neighbors are another room, a wall, or a door to another room.
- The classes Room, Door, and Wall define the components of the maze used in all our examples. We define only the parts of these classes that are important for creating a maze.
- We'll ignore players, operations for displaying and wandering around in a maze, and other important functionality that isn't relevant to building the maze. The following diagram shows the relationships between these classes:



## 10. Factory Method

**Explain in detail on factory method.(April/May 2011,2017, Nov/Dec 2013, 2015)**

**Intent**

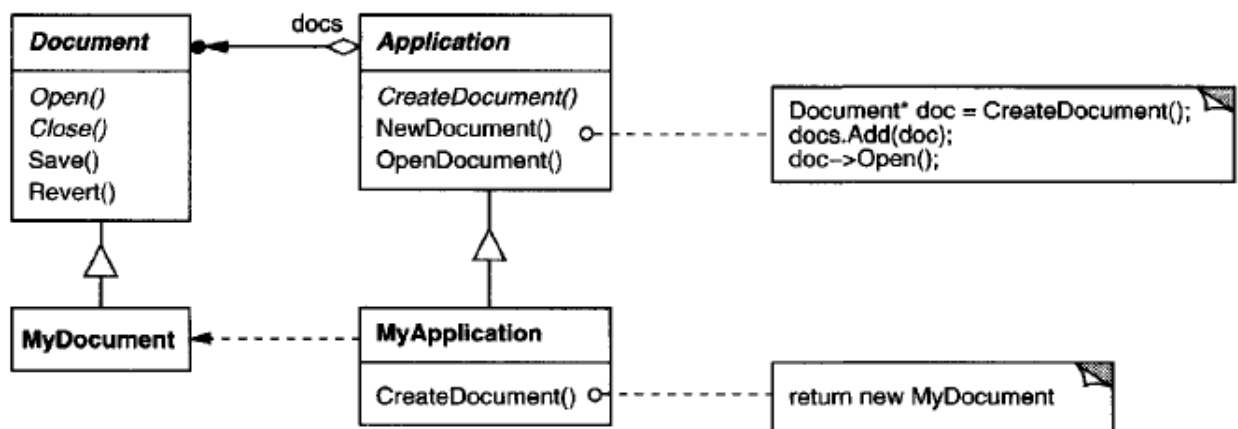
Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

**Also Known As**

Virtual Constructor

**Motivation**

- Frameworks use abstract classes to define and maintain relationships between objects.
- A framework is often responsible for creating these objects as well. Consider a framework for applications that can present multiple documents to the user.
- Two key abstractions in this framework are the classes Application and Document. Both classes are abstract, and clients have to subclass them to realize their application-specific implementations.
- To create a drawing application, for example, we define the classes DrawingApplication and DrawingDocument. The Application class is responsible for managing Documents and will create them as required—when the user selects Open or New from a menu, for example.
- Because the particular Document subclass to instantiate is application-specific, the Application class can't predict the subclass of Document to instantiate—the Application class only knows *when* a new document should be created, not *what kind* of Document to create.
- This creates a dilemma: The framework must instantiate classes, but it only knows about abstract classes, which it cannot instantiate.
- The Factory Method pattern offers a solution. It encapsulates the knowledge of which Document subclass to create and moves this knowledge out of the framework.



- Application subclasses redefine an abstract CreateDocument operation on Application to return the appropriate Document subclass.



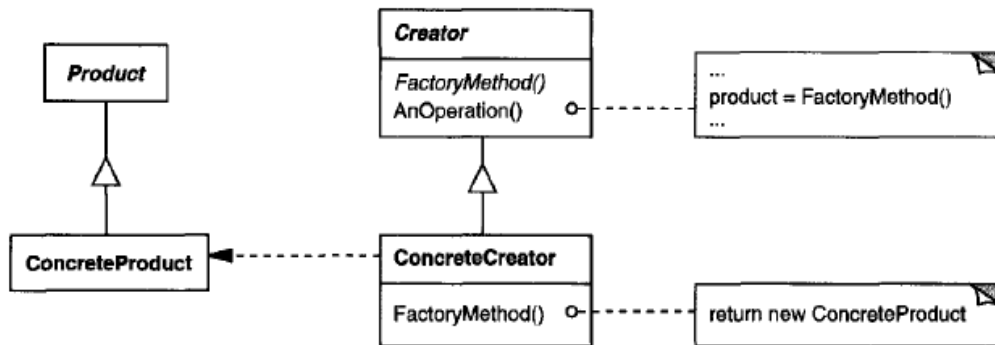
- Once an Application subclass is instantiated, it can then instantiate application-specific Documents without knowing their class. We call CreateDocument a factory method because it's responsible for "manufacturing" an object.

### Applicability

Use the Factory Method pattern when

- a class can't anticipate the class of objects it must create.
- a class wants its subclasses to specify the objects it creates.
- classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate.

### Structure



### Participants

- **Product (Document)**
  - defines the interface of objects the factory method creates.
- **ConcreteProduct (MyDocument)**
  - implements theProduct interface.
- **Creator (Application)**
  - declares the factory method, which returns an object of type Product. Creator may also define a default implementation of the factory method that returns a default Concrete Product object.
- **Concrete Creator (My Application)**
  - overrides the factory method to return an instance of aConcreteProduct.

### Collaborations

- Creator relies on its subclasses to define the factory method so that it returns an instance of the appropriate ConcreteProduct.

### Consequences

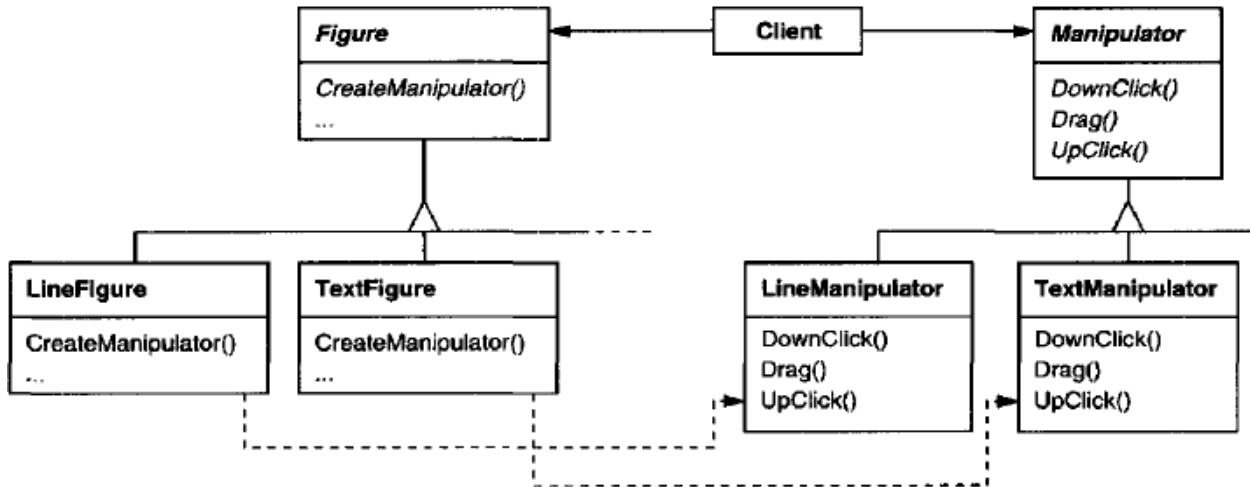
- Factory methods eliminate the need to bind application-specific classes into your code.
- The code only deals with the Product interface; therefore it can work with any user-defined Concrete Product classes.
- A potential disadvantage of factory methods is that clients might have to subclass the Creator class just to create a particular Concrete Product object.
- Here are two additional consequences of the Factory Method pattern:



1. *Provides hooks for subclasses.*

2. *Connects parallel class hierarchies.*

The resulting Manipulator class hierarchy parallels (at least partially) the Figure class hierarchy:



### Related Patterns

- Abstract Factory is often implemented with factory methods.
- Factory methods are usually called within Template Methods.
- Prototypes don't require subclassing Creator. However, they often require an Initialize operation on the Product class. Creator uses Initialize to initialize the object. Factory Method doesn't require such an operation.

## 11. Structural Patterns

### Explain in detail on structural pattern.(Nov/Dec 2015, May/June 2016)

- Structural patterns are concerned with how classes and objects are composed to form larger structures. Structural *class* patterns use inheritance to compose interfaces or implementations.
  - As a simple example, consider how multiple inheritance mixes two or more classes into one. The result is a class that combines the properties of its parent classes.
    - This pattern is particularly useful for making independently developed class libraries work together. Another example is the class form of the Adapter pattern.
      - In general, an adapter makes one interface (the adaptee's) conform to another, thereby providing a uniform abstraction of different interfaces.
        - A class adapter accomplishes this by inheriting privately from an adaptee class. The adapter then expresses its interface in terms of the adaptee's.
        - Rather than composing interfaces or implementations, structural *object* patterns describe ways to compose objects to realize new functionality.

- The added flexibility of object composition comes from the ability to change the composition at run-time, which is impossible with static class composition.
- Composite is an example of a structural object pattern. It describes how to build a class hierarchy made up of classes for two kinds of objects: primitive and composite. The composite objects let you compose primitive and other composite objects into arbitrarily complex structures.
- In the **Proxy pattern**, a proxy acts as a convenient surrogate or placeholder for another object.
- A proxy can be used in many ways. It can act as a local representative for an object in a remote address space. It can represent a large object that should be loaded on demand. It might protect access to a sensitive object.
- Proxies provide a level of indirection to specific properties of objects. Hence they can restrict, enhance, or alter these properties.
- The **Flyweight pattern** defines a structure for sharing objects. Objects are shared for at least two reasons: efficiency and consistency. Flyweight focuses on sharing for space efficiency.
- Applications that use lots of objects must pay careful attention to the cost of each object. Substantial savings can be had by sharing objects instead of replicating them. But objects can be shared only if they don't define context-dependent state.
- Flyweight objects have no such state. Any additional information they need to perform their task is passed to them when needed.
- With no context-dependent state, Flyweight objects may be shared freely. Whereas Flyweight shows how to make lots of little objects, Facade shows how to make a single object represent an entire subsystem.
- A **facade** is a representative for a set of objects. The facade carries out its responsibilities by forwarding messages to the objects it represents. The Bridge pattern separates an object's abstraction from its implementation so that you can vary them independently.
- **Decorator** describes how to add responsibilities to objects dynamically. Decorator is a structural pattern that composes objects recursively to allow an open-ended number of additional responsibilities.

#### **Related Patterns**

- Bridge has a structure similar to an object adapter, but Bridge has a different intent:
- Decorator enhances another object without changing its interface. A decorator is thus more transparent to the application than an adapter is.
- Proxy defines a representative or surrogate for another object and does not change its interface.

## 12. ADAPTER

Explain in detail on Adapter pattern.(April/May 2011, Nov/Dec 2013,2015, May/June 2016)

### **Intent**

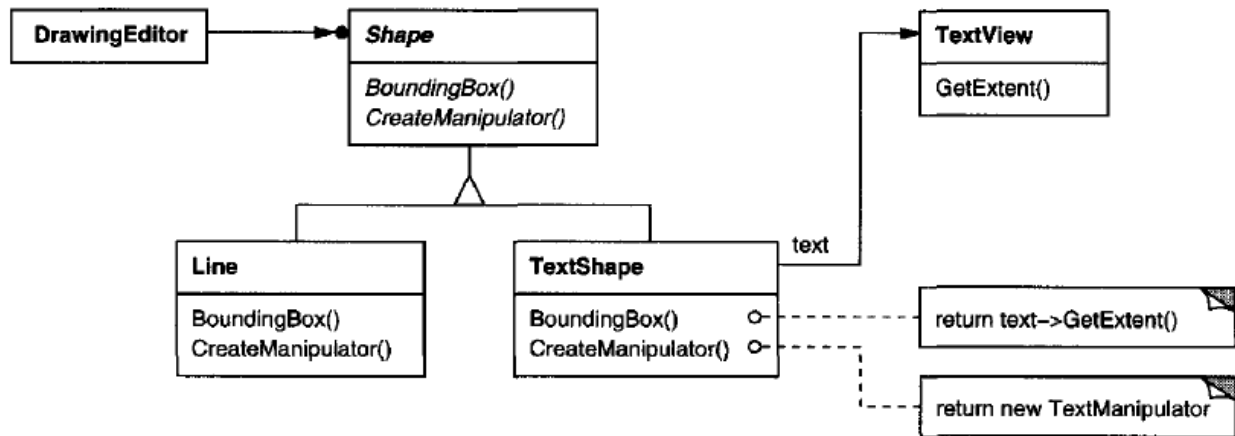
Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

### **Also Known As**

Wrapper

### **Motivation**

- Sometimes a toolkit class that's designed for reuse isn't reusable only because its interface doesn't match the domain-specific interface an application requires.
- Consider for example a drawing editor that lets users draw and arrange graphical elements (lines, polygons, text, etc.) into pictures and diagrams.
- The drawing editor's key abstraction is the graphical object, which has an editable shape and can draw itself.
- The interface for graphical objects is defined by an abstract class called Shape. The editor defines a subclass of Shape for each kind of graphical object: a Line Shape class for lines, a Polygon Shape class for polygons, and so forth.
- Classes for elementary geometric shapes like Line Shape and Polygon Shape are rather easy to implement, because their drawing and editing capabilities are inherently limited. But a Text Shape subclass that can display and edit text is considerably more difficult to implement, since even basic text editing involves complicated screen update and buffer management.
- Meanwhile, an off-the-shelf user interface toolkit might already provide a sophisticated Text View class for displaying and editing text. Ideally we'd like to reuse Text View to implement Text Shape, but the toolkit wasn't designed with Shape classes in mind. So we can't use Text View and Shape objects interchangeably.
- Even if we did, it wouldn't make sense to change Text View; the toolkit shouldn't have to adopt domain-specific interfaces just to make one application work.
- Instead, we could define Text Shape so that it *adapts* the Text View interface to Shape's. We can do this in one of two ways:
  - (1) By inheriting Shape's interface and Text View's implementation or
  - (2) By composing a Text View instance within a TextShape and implementing Text Shape and terms of textView's interface.
- These two approaches correspond to the class and object versions of the Adapterpattern. We call TextShape an **adapter**.



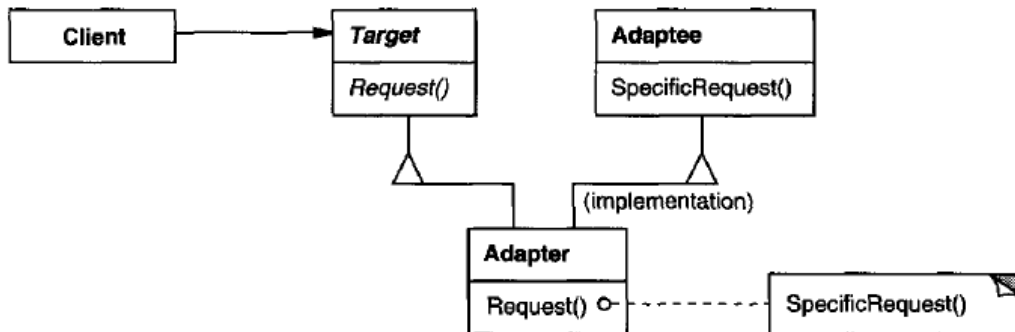
### Applicability

Use the Adapter pattern when

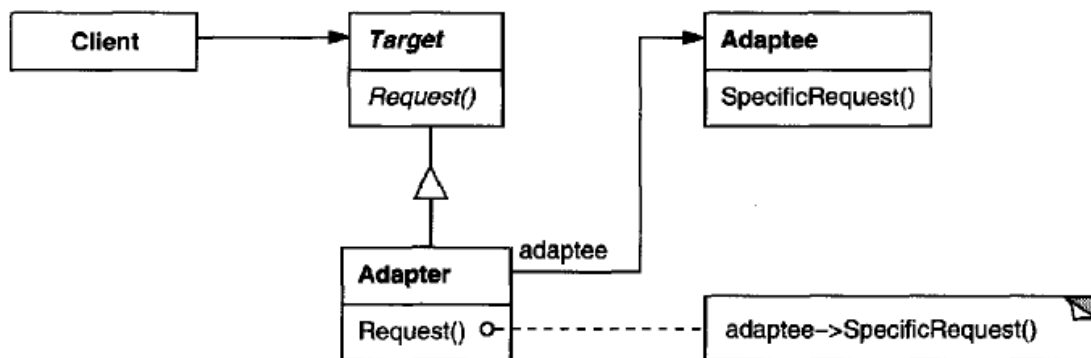
- you want to use an existing class, and its interface does not match the one you need.
- you want to create a reusable class that cooperates with unrelated or unforeseen classes, that is, classes that don't necessarily have compatible interfaces.
- *(object adapter only)* you need to use several existing subclasses, but it's unpractical to adapt their interface by subclassing every one. An object adapter can adapt the interface of its parent class.

### Structure

A class adapter uses multiple inheritance to adapt one interface to another:



An object adapter relies on object composition:





### Participants

- **Target (Shape)**  
- defines the domain-specific interface that Client uses.
- **Client (Drawing Editor)**  
- collaborates with objects conforming to the Target interface.
- **Adaptec (Text View)**  
- defines a existing interface that needs adapting.
- **Adapter (Text Shape)**  
- adapts the interface of Adaptee to the Target interface.

### Related Patterns

- Bridge has a structure similar to an object adapter, but Bridge has a different intent
- Decorator enhances another object without changing its interface. A decorator is thus more transparent to the application than an adapter is. As a consequence, Decorator supports recursive composition, which isn't possible with pure adapters.
- Proxy defines a representative or surrogate for another object and does not change its interface.

---

## 13. BRIDGE

### Explain in detail on Bridge pattern.(Nov/Dec 2015, May/June 2016)

#### Intent

Decouple an abstraction from its implementation so that the two can vary independently.

#### Also Known As

Handle/Body

#### Motivation

- When an abstraction can have one of several possible implementations, the usual way to accommodate them is to use inheritance.
- An abstract class defines the interface to the abstraction, and concrete subclasses implement it in different ways.
- But this approach isn't always flexible enough. Inheritance binds an implementation to the abstraction permanently, which makes it difficult to modify, extend, and reuse abstractions and implementations independently.

#### Applicability

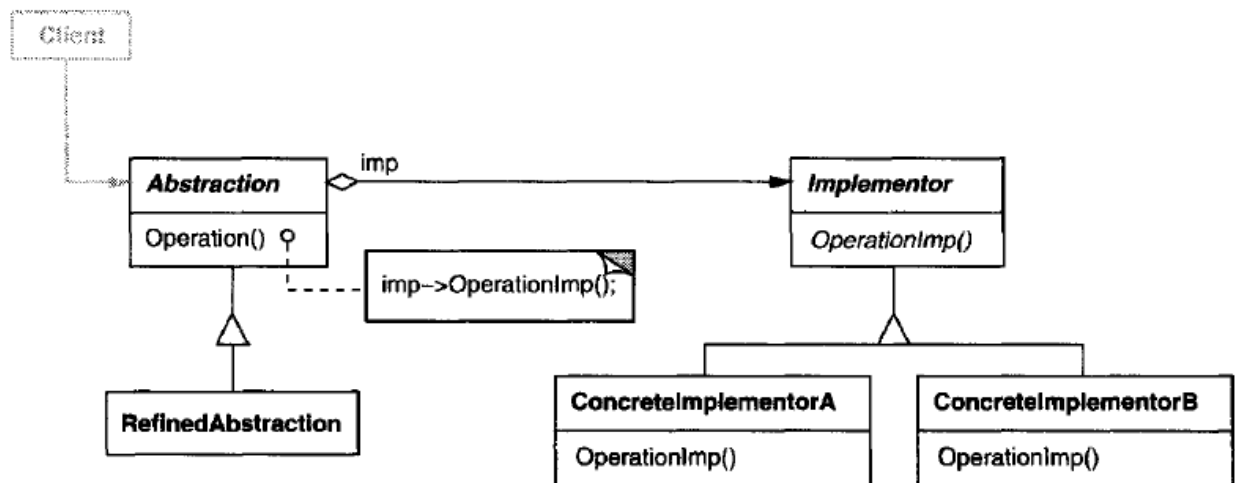
Use the Bridge pattern when

- You want to avoid a permanent binding between an abstraction and its implementation.



- Both the abstractions and their implementations should be extensible by subclassing. In this case, the Bridge pattern lets you combine the different abstractions and implementations and extend them independently.
- Changes in the implementation of an abstraction should have no impact on clients; that is, their code should not have to be recompiled.
- You have a proliferation of classes as shown earlier in the first Motivation diagram. Such a class hierarchy indicates the need for splitting an object into two parts
- You want to share an implementation among multiple objects (perhaps using reference counting), and this fact should be hidden from the client.

### Structure



### Participants

- **Abstraction**(Window)
  - defines the abstraction's interface.
  - maintains a reference to an object of type Implementer.
- **Refmed Abstraction** (Icon Window)
  - Extends the interface defined by Abstraction.
- **Implementor** (WindowImp)
  - defines the interface for implementation classes.
- **ConcreteImplementor** (XWindowImp, PMWindowImp)
  - implements the Implementor interface and defines its concrete implementation.

### Consequences

The Bridge pattern has the following consequences:

1. Decoupling interface and implementation.
2. Improved extensibility.
3. Hiding implementation details from clients.

### Related Patterns

- An AbstractFactory can create and configure a particular Bridge.
- The Adapter pattern is geared toward making unrelated classes work together.

## 14. Behavioral Patterns

**Explain in detail on Behavioral pattern.(Nov/Dec 2015)**

- Behavioral patterns are concerned with algorithms and the assignment of responsibilities between objects. Behavioral patterns describe not just patterns of objects or classes but also the patterns of communication between them.
- These patterns characterize complex control flow that's difficult to follow at run-time. They shift your focus away from flow of control to let you concentrate just on the way objects are interconnected.
- Behavioral class patterns use inheritance to distribute behavior between classes. This chapter includes two such patterns. Template Method is the simpler and more common of the two.
- A **template method** is an abstract definition of an algorithm. It defines the algorithm step by step. Each step invokes either an abstract operation or a primitive operation. A subclass fleshes out the algorithm by defining the abstract operations.
- The other behavioral class pattern is Interpreter, which represents a grammar as a class hierarchy and implements an interpreter as an operation on instances of these classes.
- Behavioral object patterns use object composition rather than inheritance. Some describe how a group of peer objects cooperate to perform a task that no single object can carry out by itself.
- An important issue here is how peer objects know about each other. Peers could maintain explicit references to each other, but that would increase their coupling. In the extreme, every object would know about every other.
- The **Mediator pattern** avoids this by introducing a mediator object between peers. The mediator provides the indirection needed for loose coupling.
- The **Observer pattern** defines and maintains a dependency between objects. The classic example of Observer is in Smalltalk Model/View/Controller, where all views of the model are notified whenever the model's state changes.
- The **Strategy pattern** encapsulates an algorithm in an object. Strategy makes it easy to specify and change the algorithm an object uses.
- The **Command pattern** encapsulates a request in an object so that it can be passed as a parameter, stored on a history list, or manipulated in other ways.
- The State pattern encapsulates the states of an object so that the object can change its behavior when its state objects changes.

**Related patterns**

- Chain of Responsibility is often applied in conjunction with Composite. There, a component's parent can act as its successor.

**15. STRATEGY**

**Explain in detail on strategy pattern.(Nov/Dec 2015, May/June 2016)**

**Intent**

Define a family of algorithms, encapsulate each one, and make them interchangeable.  
Strategy lets the algorithm vary independently from clients that use it.

### Also Known As

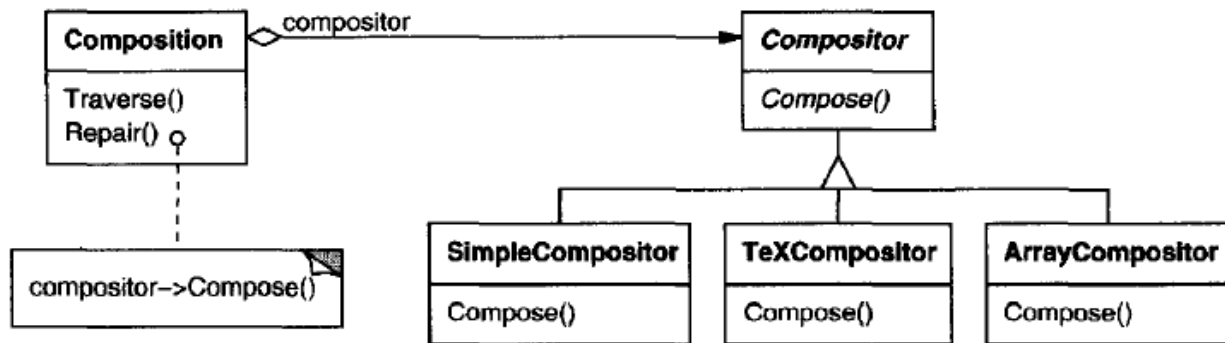
Policy

### Motivation

Many algorithms exist for breaking a stream of text in to lines. Hard-wiring all such algorithms into the classes that require them isn't desirable for several reasons:

- Clients that need linebreaking get more complex if they include the linebreaking code. That makes clients bigger and harder to maintain, especially if they support multiple linebreaking algorithms.

- Different algorithms will be appropriate at different times . We don't want to support multiple linebreaking algorithms if we don't use them all.



### Compositor subclasses implement different strategies:

- SimpleCompositor implements a simple strategy that determines linebreaks one at a time.
- TeXCompositor implements the TgX algorithm for finding linebreaks. This strategy tries to optimize line breaks globally, that is, one paragraph at a time.
- ArrayCompositor implements a strategy that selects breaks so that each row has a fixed number of items. It's useful for breaking a collection of icons into rows, for example.

A Composition maintains a reference to a Compositor object. Whenever a Composition reformats its text, it forwards this responsibility to its Compositor object. The client of Composition specifies which Compositor should be used by installing the Compositor it desires into the Composition.

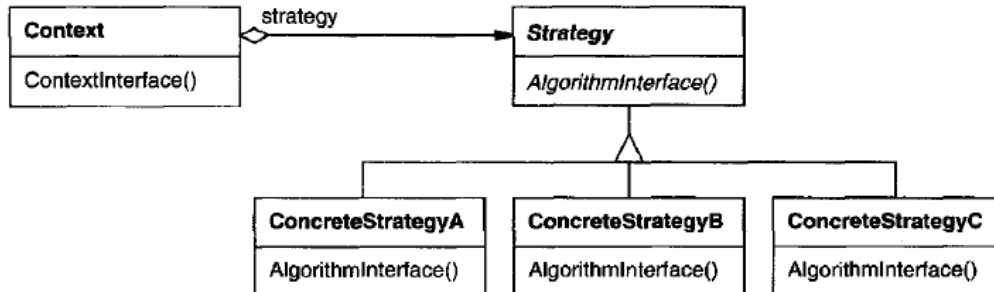
### Applicability

Use the Strategy pattern when

- Many related classes differ only in their behavior. Strategies provide a way to configure a class with one of many behaviors.
- You need different variants of an algorithm. For example, you might define algorithms reflecting different space/time trade-offs. Strategies can be used when these variants are implemented as a class hierarchy of algorithms.
- An algorithm uses data that clients shouldn't know about. Use the Strategy pattern to avoid exposing complex, algorithm-specific data structures.

- A class defines many behaviors, and these appear as multiple conditional statements in its operations. Instead of many conditionals, move related conditional branches into their own Strategy class.

### Structure



### Participants

- **Strategy (Compositor)**
  - declares an interface common to all supported algorithms. Context uses this interface to call the algorithm defined by ConcreteStrategy.
- **ConcreteStrategy (SimpleCompositor, TeXCompositor, ArrayCompositor)**
  - implements the algorithm using the Strategy interface.
- **Context (Composition)**
  - is configured with a ConcreteStrategy object.
  - maintains a reference to a Strategy object.
  - may define an interface that lets Strategy access its data.

### Consequences

The Strategy pattern has the following benefits and drawbacks:

1. Families of related algorithms.
2. An alternative to subclassing.
3. Strategies eliminate conditional statements.
4. A choice of implementations.
5. Clients must be aware of different Strategies.
6. Communication overhead between Strategy and Context.
7. Increased number of objects.

### Related Patterns

Flyweight: Strategy objects often make good flyweights.

## 16. OBSERVER

Explain in detail on observer pattern.(April/May 2011, Nov/Dec 2013, 2015, May/June 2016)

### Intent

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

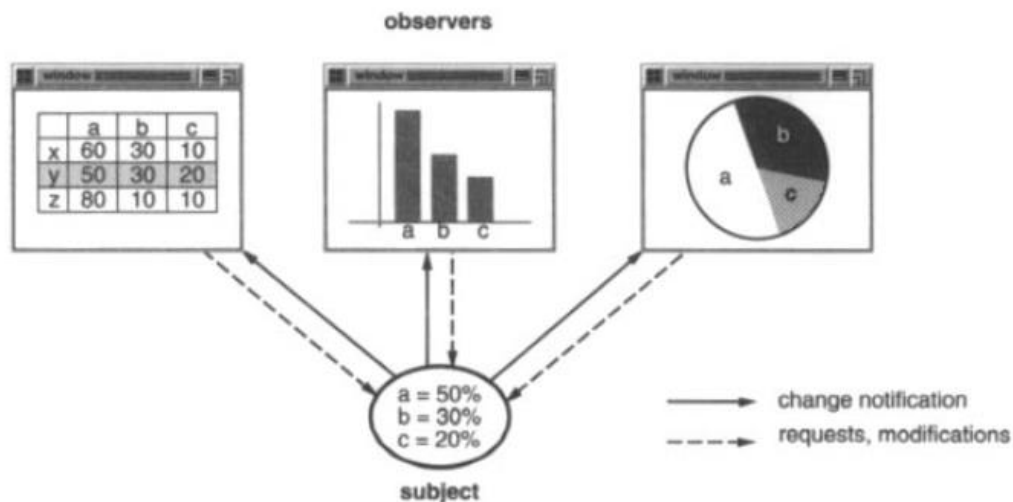
### Also Known As



Dependents, Publish-Subscribe

### Motivation

- A common side-effect of partitioning a system into a collection of cooperating classes is the need to maintain consistency between related objects. You don't want to achieve consistency by making the classes tightly coupled, because that reduces their reusability.
- For example, many graphical user interface toolkits separate the presentational aspects of the user interface from the underlying application data. Classes defining application data and presentations can be reused independently.
- They can work together, too. Both a spreadsheet object and bar chart object can depict information in the same application data object using different presentations.
- The spreadsheet and the bar chart don't know about each other, thereby letting you reuse only the one you need. But they *behave* as though they do.
- When the user changes the information in the spreadsheet, the bar chart reflects the changes immediately and vice versa.



- This behavior implies that the spreadsheet and bar chart are dependent on the data object and therefore should be notified of any change in its state. And there's no reason to limit the number of dependent objects to two; there may be any number of different user interfaces to the same data.
- The Observer pattern describes how to establish these relationships.
- The key objects in this pattern are subject and observer. A subject may have any number of dependent observers.
- All observers are notified whenever the subject undergoes a change in state. In response, each observer will query the subject to synchronize its state with the subject's state.
- This kind of interaction is also known as publish-subscribe. The subject is the publisher of notifications. It sends out these notifications without having to know who its observers are. Any number of observers can subscribe to receive notifications.

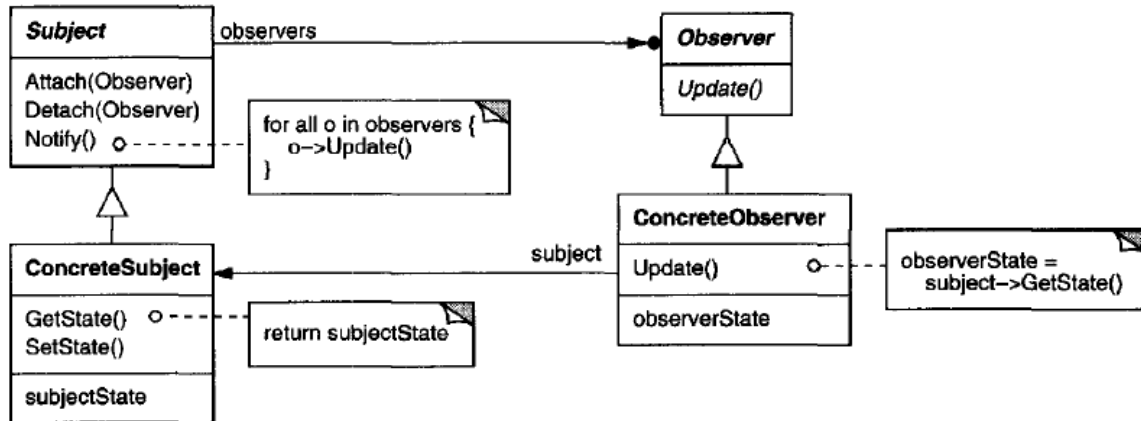
### Applicability



Use the Observer pattern in any of the following situations:

- When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently.
- When a change to one object requires changing others, and you don't know how many objects need to be changed.
- When an object should be able to notify other objects without making assumptions about who these objects are. In other words, you don't want these objects tightly coupled.

### Structure



### Participants

- **Subject**
  - knows its observers. Any number of Observer objects may observe a subject.
  - provides an interface for attaching and detaching Observer objects.
- **Observer**
  - defines an updating interface for objects that should be notified of changes in a subject.
- **ConcreteSubject**
  - stores state of interest to ConcreteObserver objects.
  - sends a notification to its observers when its state changes.
- **ConcreteObserver**
  - Maintains a reference to a ConcreteSubject object.
  - Stores state that should stay consistent with the subject's.

### Consequences

Further benefits and liabilities of the Observer pattern include the following:

1. Abstract coupling between Subject and Observer.
2. Support for broadcast communication.
3. Unexpected updates.

### Related Patterns

- Mediator
- Singleton

**CS6502- OBJECT ORIENTED ANALYSIS AND DESIGN**  
**UNIT III - CASE STUDY**

Case study – the Next Gen POS system, Inception -Use case Modeling - Relating Use cases – include, extend and generalization - Elaboration - Domain Models - Finding conceptual classes and description classes – Associations – Attributes – Domain model refinement – Finding conceptual class Hierarchies - Aggregation and Composition.

---

**1. NEXTPOS SYSTEM**

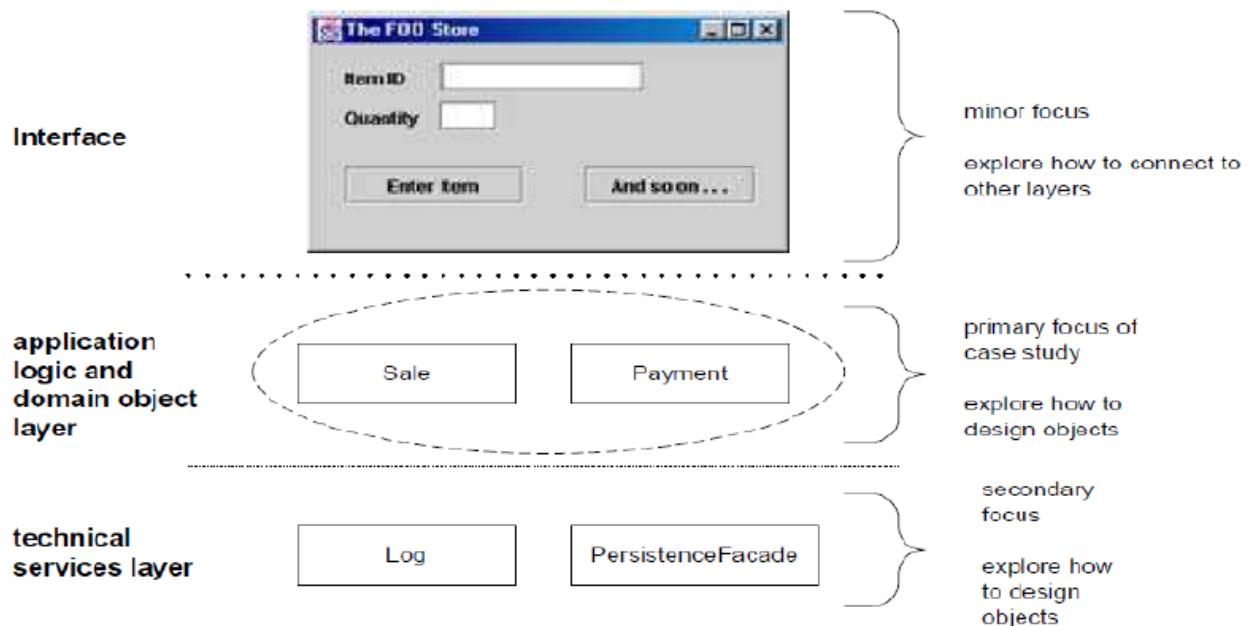
**By considering NextGen POS system perform the object oriented system development.**

- The case study is the NextGen point-of-sale (POS) system. In this apparently straight forward problem domain, we shall see that there are very interesting requirement and design problems to solve.
- In addition, it is a realistic problem; organizations really do write POS systems using object technologies.
- A POS system is a computerized application used (in part) to record sales and handle payments; it is typically used in a retail store. It includes hardware components such as a computer and bar code scanner, and software to run the system.
- It interfaces to various service applications, such as a third-party tax calculator and inventory control. These systems must be relatively fault-tolerant; that is, even if remote services are temporarily unavailable (such as the inventory system), they must still be capable of capturing sales and handling at least cash payments (so that the business is not crippled).
- A POS system increasingly must support multiple and varied client-side terminals and interfaces. These include a thin-client Web browser terminal, a regular personal computer with something like a Java Swing graphical user interface, touch screen input, wireless PDAs, and so forth.
- Furthermore, we are creating a commercial POS system that we will sell to different clients with disparate needs in terms of business rule processing.
- Each client will desire a unique set of logic to execute at certain predictable points in scenarios of using the system, such as when a new sale is initiated or when a new line item is added. Therefore, we will need a mechanism to provide this flexibility and customization.
- Using an iterative development strategy, we are going to proceed through requirements, object-oriented analysis, design, and Implementation.

**1.1 Architectural Layers and Case Study Emphasis**

- A typical object-oriented information system is designed in terms of several architectural layers or subsystems.
- The following is not a complete list, but provides an example:
  - **User Interface:** Graphical interface; windows.
  - **Application Logic and Domain Objects:** Software objects representing domain concepts (for example, a software class named Sale) that fulfill application requirements.

- **Technical Services:** General purpose objects and subsystems that provide supporting technical services, such as interfacing with a database or error logging. These services are usually application-independent and reusable across several systems.



- OOA/D is generally most relevant for modeling the application logic and technical service layers.
- The NextGen case study primarily emphasizes the problem domain objects, allocating responsibilities to them to fulfill the requirements of the application. Object-oriented design is also applied to create a technical service subsystem for interfacing with a database.
- In this design approach, the UI layer has very little responsibility; it is said to be thin. Windows do not contain code that performs application logic or processing.
- Rather, task requests are forwarded on to other layers.

## 2. INCEPTION PHASE

### Explain briefly about Inception Phase. (Nov/Dec 2015)

- This is the part of the project where the original idea is developed. The amount of work done here is dependent on how formal project planning is done in your organization and the size of the project.
- During this part of the project some technical risk may be partially evaluated and/or eliminated. This may be done by using a few throw away prototypes to test for technical feasibility of specific system functions.
- Normally this phase would take between two to six weeks for large projects and may be only a few days for smaller projects.
- The following should be done during this phase:
  1. Project idea is developed.

2. Assess the capabilities of any current system that provides similar functionality to the new project even if the current system is a manual system. This will help determine cost savings that the new system can provide.

3. Utilize as many users and potential users as possible along with technical staff, customers, and management to determine desired system features, functional capabilities, and performance requirements. Analyze the scope of the proposed system.

4. Identify feature and functional priorities along with preliminary risk assessment of each system feature or function.

5. Identify systems and people the system will interact with.

6. For large systems, break the system down into subsystems if possible.

7. Identify all major use cases and describe significant use cases. No need to make expanded use cases at this time. This is just to help identify and present system functionality.

8. Develop a throw away prototype of the system with breadth and not depth. This prototype will address some of the greatest technical risks. The time to develop this prototype should be specifically limited. For a project that will take about one year, the prototype should take one month.

9. Present a business case for the project (white paper) identifying rough cost and value of the project. The white paper is optional for smaller projects. Define goals, estimate risks, and resources required to complete the project.

10. Set up some major project milestones (mainly for the elaboration phase). A rough estimate of the overall project size is made.

11. Preliminary determination of iterations and requirements for each iteration. This outlines system functions and features to be included in each iteration. Keep in mind that this plan will likely be changes as risks are further assessed and more requirements are determined.

12. Management Approval for a more serious evaluation of the project. This phase is done once the business case is presented with major milestones determined (not cast in stone yet) and management approves the plan.

At this point the following should be complete:

- Business case (if required) with risk assessment.
- Preliminary project plan with preliminary iterations planned.
- Core project requirements are defined on paper.
- Major use cases are defined.

The inception phase has only one iteration. All other phases may have multiple iterations. The overriding goal of the inception phase is to achieve concurrence among all stakeholders on the lifecycle objectives for the project.

The inception phase is of significance primarily for new development efforts, in which there are significant business and requirements risks which must be addressed before the project can proceed.

For projects focused on enhancements to an existing system, the inception phase is more brief, but is still focused on ensuring that the project is both worth doing and possible to do.

## 2.1 Objectives:

The primary objectives of the Inception phase include:

- Establishing the project's software scope and boundary conditions, including an operational vision, acceptance criteria and what is intended to be in the product and what is not.



- Discriminating the critical use cases of the system, the primary scenarios of operation that will drive the major design tradeoffs. Exhibiting, and maybe demonstrating, at least one candidate architecture against some of the primary scenarios.
- Estimating the overall cost and schedule for the entire project (and more detailed estimates for the elaboration phase that will immediately follow) Estimating potential risks (the sources of unpredictability)
- Preparing the supporting environment for the project.

## 2.2 Essential Activities:

The essential activities of the Inception include:

- **Formulating the scope of the project.** This involves capturing the context and the most important requirements and constraints to such an extent that you can derive acceptance criteria for the end product.
  - **Planning and preparing a business case.** Evaluating alternatives for risk management, staffing, project plan, and cost/schedule/profitability tradeoffs.
  - **Synthesizing candidate architecture,** evaluating tradeoffs in design, and in make/buy/reuse, so that cost, schedule and resources can be estimated. The aim here is to demonstrate feasibility through some kind of proof of concept. This may take the form of a model which simulates what is required, or an initial prototype which explores that are considered to be the areas of high risk.
  - **Preparing the environment for the project,** assessing the project and the organization, selecting tools, deciding which parts of the process to improve.

## 2.3 Inception Phase includes:

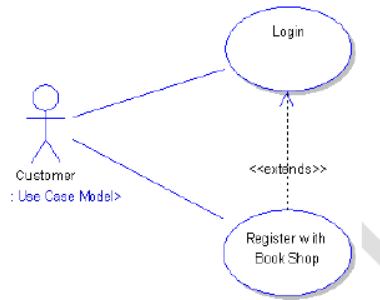
- Refining the scope of the project
- Project planning
- Risk identification and analysis
- Preparing the project environment
- Estimating the Budget

## 3. USECASE MODELING:

**Explain with an example, how use case modeling is used to describe functional requirements? Identify the actors, scenario and use cases for the example.**

- The Use Case Model describes the proposed functionality of the new system. A Use Case represents a discrete unit of interaction between a user (human or machine) and the system.
- A Use Case is a single unit of meaningful work; for example login to system, register with system and create order are all Use Cases.
- Each Use Case has a description which describes the functionality that will be built in the proposed system.
- A Use Case may 'include' another Use Case's functionality or 'extend' another Use Case with its own behavior.
- Use Cases are typically related to 'actors'. An actor is a human or machine entity that interacts with the system to perform meaningful work.

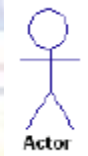




### 3.1 Actors:

An Actor is a user of the system. This includes both human users and other computer systems. An Actor uses a Use Case to perform some piece of work which is of value to the business.

The set of Use Cases an actor has access to define their overall role in the system and the scope of their action.



### 3.2 Constraints, Requirements and Scenarios

The formal specification of a Use Case includes:

#### 1. Requirements:

These are the formal functional requirements that a Use Case must provide to the end user. They correspond to the functional specifications found in structured methodologies. A requirement is a contract that the Use Case will perform some action or provide some value to the system.

#### 2. Constraints:

These are the formal rules and limitations that a Use Case operates under, and includes prepost- and invariant conditions. A pre-condition specifies what must have already occurred or be in place before the Use Case may start. A post-condition documents what will be true once the Use Case is complete. An invariant specifies what will be true throughout the time the Use Case operates.

#### 3. Scenarios:

Scenarios are formal descriptions of the flow of events that occurs during a Use Case instance. These are usually described in text and correspond to a textual representation of the Sequence Diagram.

### 3.3 USE CASE RELATIONSHIPS

Use case relationships is divided into three types

1. Include relationship
2. Extend relationship
3. Generalization

#### 1. Include relationship:

- It is common to have some practical behavior that is common across several use cases.
- It is simply to underline it or highlight it in some fashion  
Example: Paying by credit: Include Handle Credit Payment

## 2. Extend relationship:

- Extending the use case or adding new use case to the process Extending use case is triggered by some conditions called extension point.

## 3. Generalization:

- Complicated work and unproductive time is spending in this use case relationship. Use case experts are successfully doing use case work without this relationship.

## 3.4 INCLUDES AND EXTENDS RELATIONSHIPS BETWEEN USE CASES

One Use Case may include the functionality of another as part of its normal processing. Generally, it is assumed that the included Use Case will be called every time the basic path is run.

An example may be to list a set of customer orders to choose from before modifying a selected order in this case the <list orders> Use Case may be included every time the <modify order> Use Case is run.

A Use Case may be included by one or more Use Cases, so it helps to reduce duplication of functionality by factoring out common behavior into Use Cases that are re-used many times. One Use Case may extend the behavior of another - typically when exceptional circumstances are encountered.

### Relationships between Use Cases

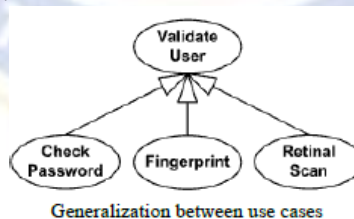
Use cases could be organized using following relationships:

- Generalization
- Association
- Extend
- Include Generalization Between Use Cases

### 3.4.1 Generalization between use cases

Generalization between use cases is similar to generalization between classes; child use case inherits properties and behavior of the parent use case and may override the behavior of the parent.

**Notation:** Generalization is rendered as a solid directed line with a large open arrowhead (same as generalization between classes).



### 3.4.2 Association between Use Cases

Use cases can only be involved in binary Associations. Two use cases specifying the same subject cannot be associated since each of them individually describes a complete usage of the system.

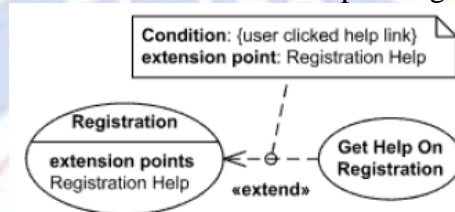
### 3.4.3 Extend Relationship

- Extend is a directed relationship from an extending use case to an extended use case that specifies how and when the behavior defined in usually supplementary (optional) extending use case can be inserted into the behavior defined in the use case to be extended.

- The extension takes place at one or more extension points defined in the extended use case. The extend relationship is owned by the extending use case. The same extending use case can extend more than one use case, and extending use case may itself be extended.
- Extend relationship between use cases is shown by a dashed arrow with an open arrowhead from the extending use case to the extended (base) use case. The arrow is labeled with the keyword <<extend>>.



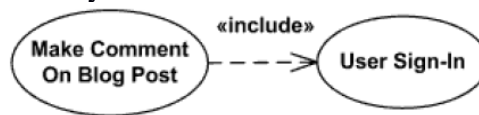
- Registration use case is meaningful on its own, and it could be extended with optional Get Help On Registration use case.
- The condition of the extend relationship as well as the references to the extension points are optionally shown in a Note attached to the corresponding extend relationship.



- Registration use case is conditionally extended by Get Help On Registration use case in extension point Registration Help

#### 3.4.4 Include Relationship

- An include relationship is a directed relationship between two use cases, implying that the behavior of the required (not optional) included use case is inserted into the behavior of the including (base) use case.
- Including use case depends on the addition of the included use case. The include relationship is intended to be used when there are common parts of the behavior of two or more use cases.
- This common part is extracted into a separate use case to be included by all the base use cases having this part in common.
- As the primary use of the include relationship is to reuse common parts, including use cases are usually not complete by themselves but dependent on the included use cases.
- Include relationship between use cases is shown by a dashed arrow with an open arrowhead from the including (base) use case to the included (common part) use case. The arrow is labeled with the keyword <<include>>.



#### **4. ELABORATION PHASE**

##### **Write briefly about elaboration. (Nov/Dec 2013, April/May 2017)**

- The primary purpose of this phase is to complete the most essential parts of the project that are high risk and plan the construction phase. This is the part of the project where technical risk is fully evaluated and/or eliminated by building the highest risk parts of the project.

- During this phase personnel requirements should be more accurately determined along with estimated man hours to complete the project. The complete cost and time frame of the project is more firmly determined.

- During this phase how the system will work must be considered. Use cases will help identify risks. **Steps to take during this phase:**

1. Complete project plan with construction iterations planned with requirements for each iteration.

2. 80% of use cases are completed. Significant use cases are described in detail.

3. The project domain model is defined. (Don't get bogged down)

4. Rank use cases by priority and risk.

Do the highest priority and highest risk use cases first. Items that may be high risk:

- Overall system architecture especially when dealing with communication between subsystems.
- Team structure.
- Anything not done before or used before such as a new programming language, or using the unified/iterative process for the first time.

5. Begin design and development of the riskiest and highest priority use cases. There will be an iteration for each high risk and priority use case.

6. Plan the iterations for the construction phase. This involves choosing the length of the iterations and deciding which use cases or parts of use cases will be implemented during each iteration. Develop the higher priority and risk use cases during the first iterations in the construction phase.

- As was done on a preliminary level in the previous phase, the value (priority) of use cases and their respective risks must be more fully assessed in this phase.

- This may be done by either assigning a number to each use case for both value and risk. or categorize them by high, medium, or low value and risk. Time required for each use case should be estimated to the man week. Do the highest priority and highest risk use cases first.

##### **Requirements to be completed for this phase include:**

- Description of the software architecture. Therefore most use cases should be done, activity diagrams, state charts, system sequence diagrams, and the domain model should be mostly complete.
- A prototype that overcomes the greatest project technical risk and has minimal high priority functionality.
- Complete project plan.



## 5. DOMAIN MODEL

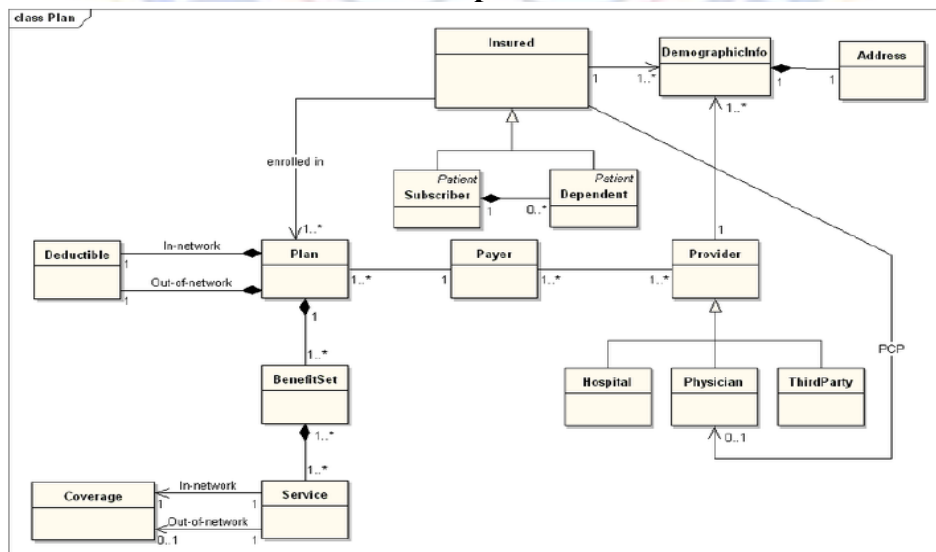
**Illustrate the concept of Domain model with examples. (April/May 2011& May/June 2016).**

A **domain model**, or **Domain Object Model (DOM)** in problem solving and software engineering can be thought of as a conceptual model of a domain of interest (often referred to as a problem domain) which describes the various entities, their attributes and relationships, plus the constraints that govern the integrity of the model elements comprising that problem domain.

### Usage

- A well-thought domain model serves as a clear depiction of the conceptual fabric of the problem domain and therefore is invaluable to ensure all stakeholders are aligned in the scope and meaning of the concepts indigenous to the problem domain.
- A high fidelity domain model can also serve as an essential input to solution implementation within a software development cycle since the model elements comprising the problem domain can serve as key inputs to code construction, whether that construction is achieved manually or through automated code generation approaches.
- It is important, however, not to compromise the richness and clarity of the business meaning depicted in the domain model by expressing it directly in a form influenced by design or implementation concerns.
- The domain model is one of the central artifacts in the project development approach called Feature Driven Development (FDD).
- In UML, a class diagram is used to represent the domain model. In Domain-driven design, the domain model (Entities and Value objects) is a part of the Domain layer which often also includes other concepts such as Services.

### Sample domain model for a health insurance plan





### Concepts: Conceptual Data Modeling

Conceptual data modeling represents the initial stage in the development of the design of the persistent data and persistent data storage for the system.

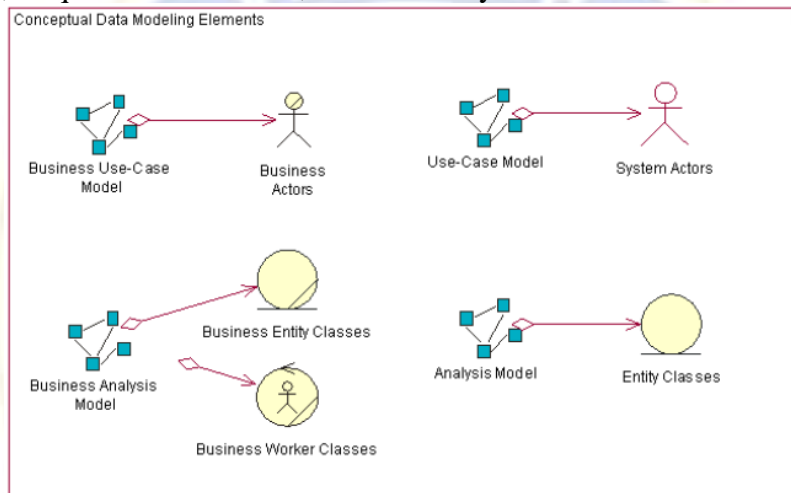
The Data Model typically evolves through the following **three general stages**:

**Conceptual**—this stage involves the identification of the high level key business and system entities and their relationships that define the scope of the problem to be addressed by the system.

**Logical**—this stage involves the refinement of the conceptual high level business and system entities into more detailed logical entities.

**Physical**—this stage involves the transformation of the logical class designs into detailed and optimized physical database table designs. The physical stage also includes the mapping of the database table designs to tables spaces and to the database component in the database storage design.

The figure below shows the set of Conceptual Data Model elements that reside in the Business Models, Requirements Models, and the Analysis Model.



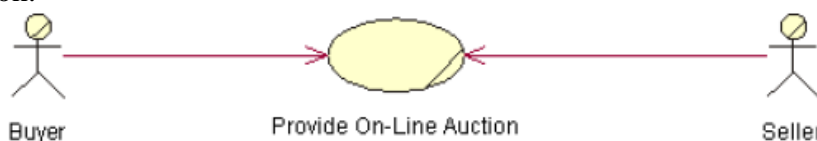
The following sections describe the elements of the Business Models, Use-Case Model, and Analysis Model that can be used to define the initial Conceptual Data Model for persistent data in the system.

### Conceptual Data Modeling Elements

#### **Business Use-Case Model**

The Business Use-Case Model consists of Business Actors and Business Use Cases. The Business Use Cases represent key business processes that are used to define the context for the system to be developed. Business Actors represent key external entities that interact with the business through the Business Use Cases.

The figure below shows a very simple example Business Use-Case Model for an online auction application.



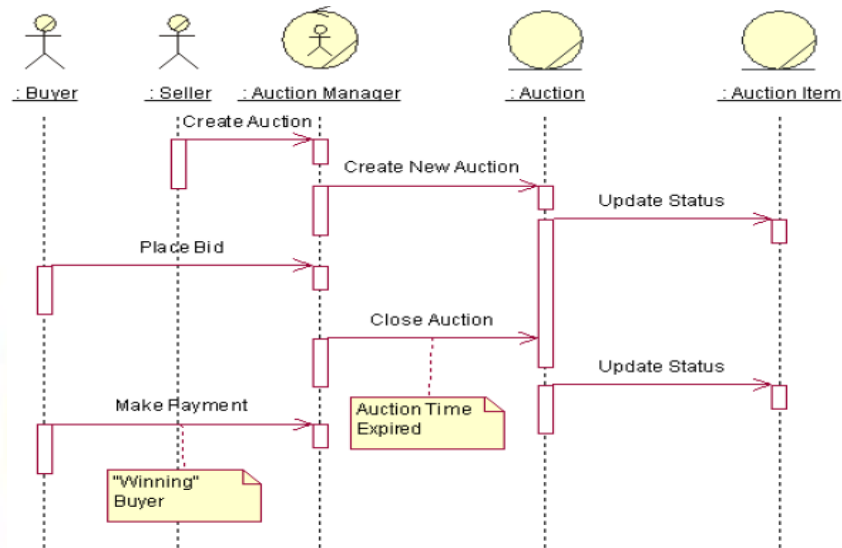
As entities of significance to the problem of space for the system, Business Actors are candidate entities for the Conceptual Data Model. In the example above, the Buyer and Seller

Business Actors are candidate entities for which the online auction application must store information.

### Business Analysis Model

The Business Analysis Model contains classes that model the Business Workers and Business Entities identified from analysis of the workflow in the Business Use Case. Business Workers represent the participating workers that perform the actions needed to carry out that workflow. Business Entities are "things" that the Business Workers use or produce during that workflow.

The figure below shows an example sequence diagram that depicts Business Workers and Business Entities from one scenario of the Business Use Case titled "Provide Online Auction" for managing an auction.



### Conceptual Class Category List

A conceptual class is a real-world concept or thing; a conceptual or essential perspective. At the noun filtering stage we are looking for conceptual classes.

#### Types of Classes

During use case realization, we identify mainly **four "types" of classes**, boundary classes, data store classes and control classes.

**The entity classes** represent the information that the system uses. Examples of entity classes are: Customer, Product, and Supplier. Entity classes are essential to the system as the expected functionality of the system includes maintaining information about them or retrieving information from them.

**The boundary classes** represent the interaction between the system and its actors. A GUI form is an example of a boundary class.

**Data store classes** encapsulate the design decisions about data storage and retrieval strategies. This provides us flexibility to move an application from database platform to another.

**The control classes** represent the control logic of the system. They implement the flow of events as given in a use case.

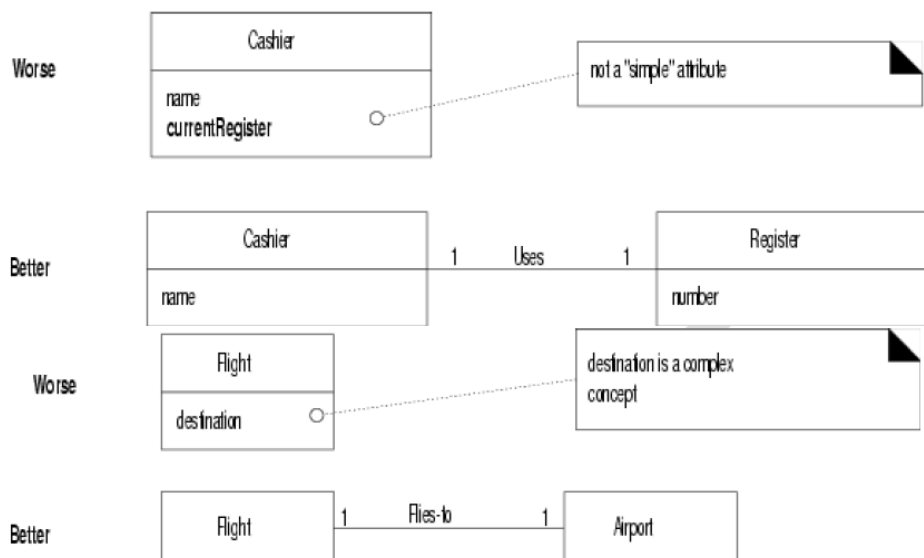
## 7. Attributes

**Attribute** - a logical data value of an object

In UML:

- Attributes are shown in the second compartment of the class box.
- The type of an attribute may optionally be shown.

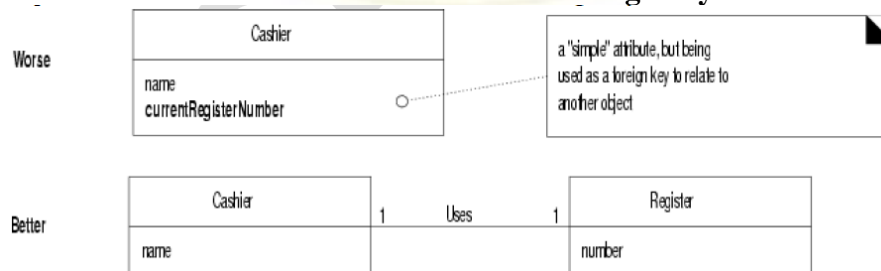
In a domain model, attributes and data types should be simple. Complex concepts should be represented by an association to another conceptual class.



An attribute should be what the UML standard calls a **data type**: a set of values for which unique identity is not meaningful. Numbers, strings, Booleans, dates, times, phone numbers, and addresses are examples of data types. Values of these types are called **value objects**.

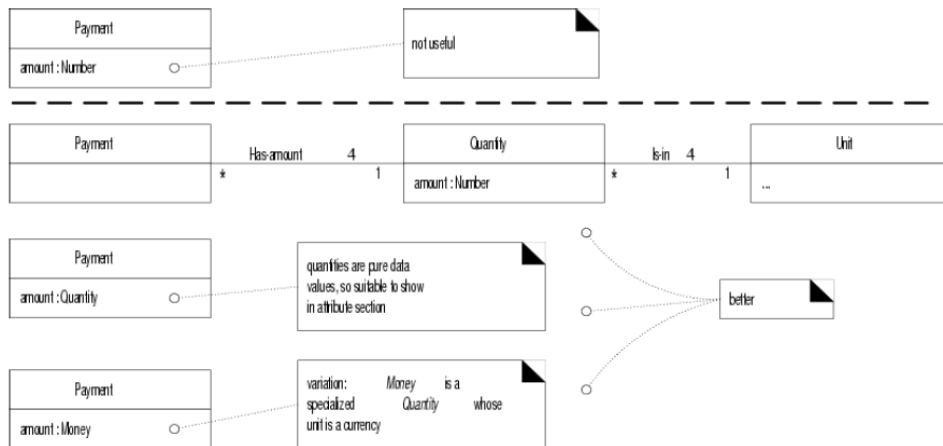
### Relating Types

Conceptual classes in a domain model should be related by associations, not attributes. In particular, an attribute should not be used as a kind of **foreign key**.



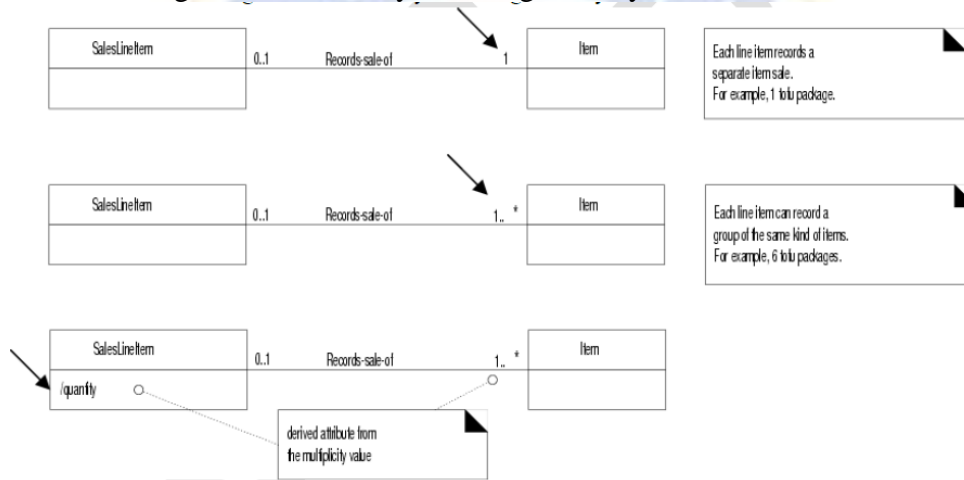
### Quantities and Units

Quantities with associated units should be represented either as conceptual classes or as attributes of specialized types that imply units (e.g., Money or Weight).

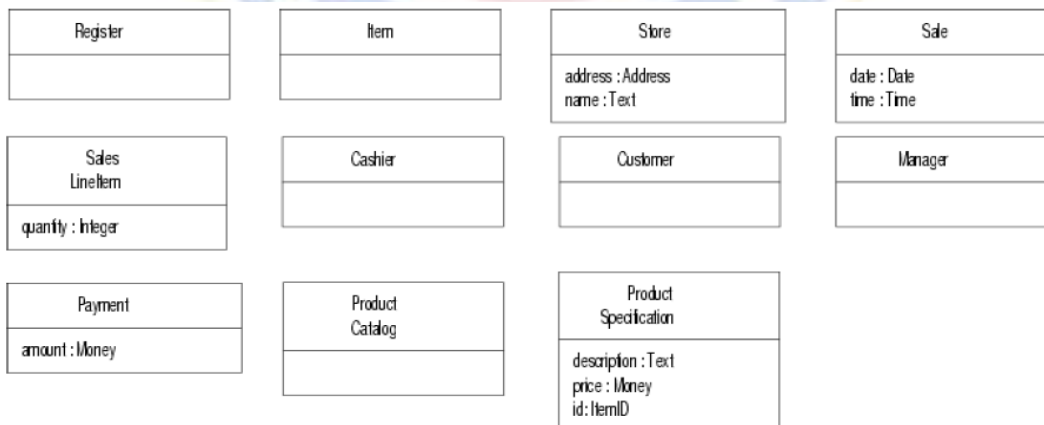


### Derived Attributes

A quantity that can be calculated from other values, such as role multiplicities, is a **derived attribute**, designated in UML by a leading slash symbol.



### NextGen POS Domain Model Attributes



## 8. Strategies to Identify Conceptual Classes

Explain the guidelines for finding conceptual classes with neat diagrams,(May/June 2016, April/May 2017)

- Two techniques are presented in the following sections:
  1. Use a conceptual class category list.
  2. Identify noun phrases.
- Another excellent technique for domain modeling is the use of **analysis patterns**, which are existing partial domain models created by experts
- Finding Conceptual Classes with Noun Phrase Identification
- Another useful technique (because of its simplicity) is linguistic analysis: identify the nouns and noun phrases in textual descriptions of a domain, and consider them as candidate conceptual classes or attributes.

Care must be applied with this method; a mechanical noun-to-class mapping isn't possible, and words in natural languages are ambiguous. Nevertheless, it is another source of inspiration. The **fully dressed use cases** are an excellent description to draw from for this analysis. For example, the current scenario of the Process Sale use case can be used.

### **Main Success Scenario (or Basic Flow):**

1. **Customer** arrives at a **POS checkout** with **goods** and/or **services** to purchase.
2. **Cashier** starts a new **sale**.
3. **Cashier** enters **item identifier**.
4. System records **sale line item** and presents **item description, price**, and running **total**. Price calculated from a set of price rules. Cashier repeats steps 2-3 until indicates done.
5. System presents total with **taxes** calculated.
6. Cashier tells Customer the total, and asks for **payment**.
7. Customer pays and System handles payment.
8. System logs the completed **sale** and sends sale and payment information to the external **Accounting** (for accounting and **commissions**) and **Inventory** systems (to update inventory).
9. System presents **receipt**.
10. Customer leaves with receipt and goods (if any).

### **Extensions (or Alternative Flows):**

- 7a. Paying by cash:
1. Cashier enters the cash **amount tendered**.
  2. System presents the **balance due**, and releases the **cash drawer**.
  3. Cashier deposits cash tendered and returns balance in cash to Customer.
  4. System records the cash payment.

The domain model is a visualization of noteworthy domain concepts and vocabulary. Thus, they are a rich source to mine via noun phrase identification.

Some of these noun phrases are candidate conceptual classes, some may refer to conceptual classes that are ignored in this iteration (for example, "Accounting" and "commissions"), and some may be attributes of conceptual classes.



A **weakness of this approach** is the imprecision of natural language; different noun phrases may represent the same conceptual class or attribute, among other ambiguities. Nevertheless, it is recommended in combination with the Conceptual Class Category List technique.

### **Specification or Description Conceptual Classes**

The following discussion may at first seem related to a rare, highly specialized issue. However, it turns out that the need for specification conceptual classes (as will be defined) is common in any domain models. Thus, it is emphasized.

Note that in earlier times a register was just one possible implementation of how to record sales. The term has acquired a generalized meaning over time.

#### **Assume the following:**

- An Item instance represents a physical item in a store; as such, it may even have a serial number.
- An Item has a description, price, and itemID, which are not recorded anywhere else.
- Everyone working in the store has amnesia.
- Every time a real physical item is sold, a corresponding software instance of Item is deleted from "software land."

With these assumptions, what happens in the following scenario?

There is strong demand for the popular new vegetarian burger—ObjectBurger. The store sells out, implying that all Item instances of ObjectBurgers are deleted from computer memory. Now, here is the heart of the problem: If someone asks, "How much do Object Burgers cost?", no one can answer, because the memory of their price was attached to inventoried instances, which were deleted as they were sold.

Notice also that the current model, if implemented in software as described, has duplicate data and is space-inefficient because the description, price, and itemID are duplicated for every Item instance of the same product.

### **The Need for Specification or Description Conceptual Classes**

- The preceding problem illustrates the need for a concept of objects that are specifications or descriptions of other things. To solve the Item problem, what is needed is a ProductSpecification (or ItemSpecification, ProductDescription) conceptual class that records information about items. A ProductSpecification does not represent an Item, it represents a description of information about items. Note that even if all inventoried items are sold and their corresponding Item software instances are deleted, the ProductSpecifications still remain.

- Description or specification objects are strongly related to the things they describe. In a domain model, it is common to state that an XSpecification Describes an X.

- The need for specification conceptual classes is common in sales and product domains. It is also common in manufacturing, where a description of a manufactured thing is required that is distinct from the thing itself. Time and space have been taken in motivating specification conceptual classes because they are very common; it is not a rare modeling concept.

## When Are Specification Conceptual Classes Required?

### The following guideline suggests when to use specifications:

Add a specification or description conceptual class (for example, ProductSpecification) when:

1. There needs to be a description about an item or service, independent of the current existence of any examples of those items or services.
2. Deleting instances of things they describe (for example, Item) results in a loss of information that needs to be maintained, due to the incorrect association of information with the deleted thing.
3. It reduces redundant or duplicated information.

## 9. Domain model Refinement

### Explain Domain model refinement with example. (Nov/Dec 2015, May/June 2016)

#### Association Classes

The following domain requirements set the stage for association classes:

- Authorization services assign a merchant ID to each store for identification during communications.
- A payment authorization request from the store to an authorization service needs the merchant ID that identifies the store to the service.
- Furthermore, a store has a different merchant ID for each service. Where in the UP Domain Model should the merchant ID attribute reside? Placing *merchantID* in *Store* is incorrect because a *Store* can have more than one value for *merchantID*. The same is true with placing it in *Authorization-Service* (see Fig1 given below).

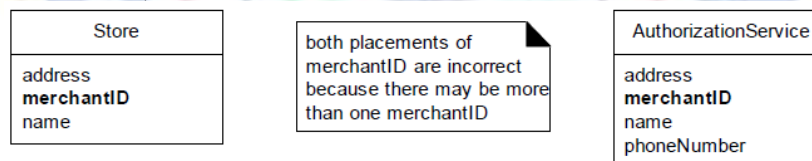


Fig1. Inappropriate use of an attribute.

#### Guidelines

Guidelines for adding association classes include the following:

Clues that an association class might be useful in a domain model:

- An attribute is related to an association.
- Instances of the association class have a life-time dependency on the association.
- There is a many-to-many association between two concepts, and information associated with the association itself.

Fig2 given below illustrates some other examples of association classes.

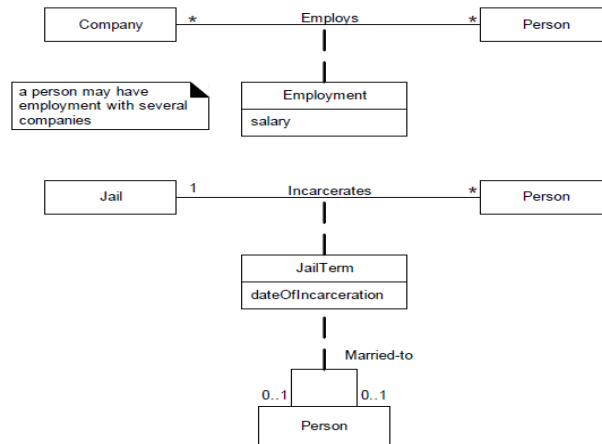


Fig2. Association classes.

**Association Role Names**

Each end of an association is a role, which has various properties, such as:

- Name
- Multiplicity

A role name identifies an end of an association and ideally describes the role played by objects in the association. Fig3 given below shows role name examples.

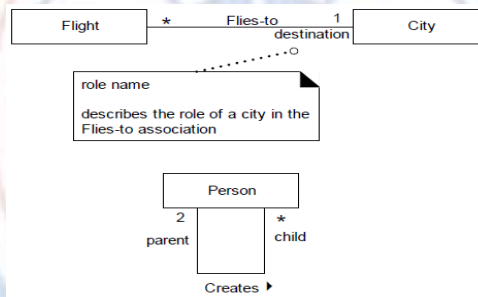


Fig3. Role names.

**Derived Elements**

A derived element can be determined from others. Attributes and associations are the most common derived elements.

For example, a *Sale total* can be derived from *SalesLineItem* and *Product-Specification* information (see Fig4 below). In the UML, it is shown with a "/" preceding the element name.

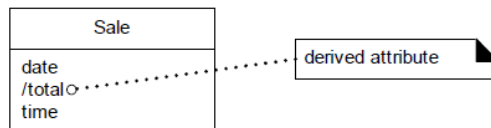


Fig4. Derived attribute.

The other approach, more robust, is to associate a collection of *ProductPrices* with a *ProductSpecification*, each with an associated applicable time interval.

Thus, the organization can record all past prices (to resolve the sale price problem, and for trend analysis) and also record future planned prices (see Fig5 below).

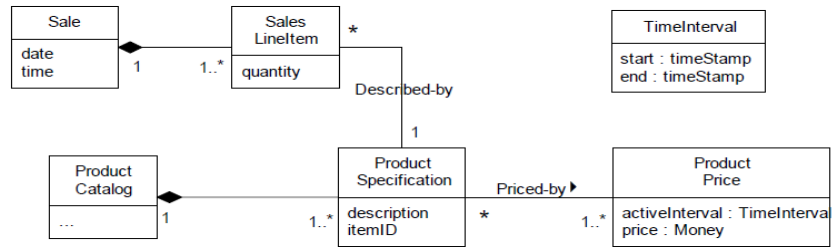


Fig5. ProductPrices and time intervals.

### Qualified Associations

A **qualifier** may be used in an association; it distinguishes the set of objects at the far end of the association based on the qualifier value. An association with a qualifier is a **qualified association**.

For example, *ProductSpecifications* may be distinguished in a *ProductCatalog* by their *itemID*, as illustrated in Fig6(b).

As contrasted in Fig6(a) vs. (b), qualification reduces the multiplicity at the far end from the qualifier, usually down from many to one. Depicting a qualifier in a domain model communicates how, in the domain, things of one class are distinguished in relation to another class.

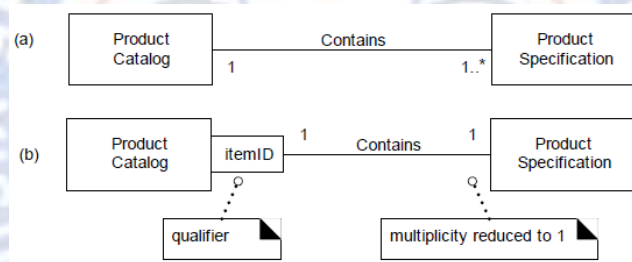


Fig6. Qualified association.

### Reflexive Associations

A concept may have an association to itself; this is known as a **reflexive association** (see Fig7 below).

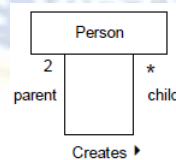
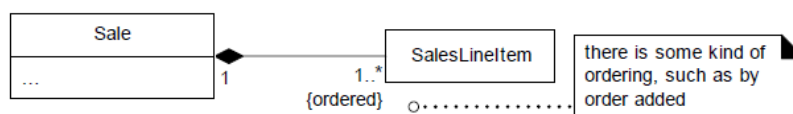


Fig7. Reflexive association.

### Ordered Elements

If associated objects are ordered, this can be shown as in figure below. For example, the *SalesLineItems* must be maintained in the order entered.



### UML Package Notation

To review, a UML package is shown as a tabbed folder (see Fig8 below). Subordinate packages may be shown within it. The package name is within the tab if the package depicts its elements; otherwise, it is centered within the folder itself.

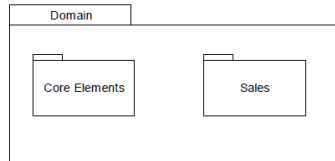


Fig8. A UML package.

### Ownership and References

An element is *owned* by the package within which it is defined, but may be *referenced* in other packages. In that case, the element name is qualified by the package name using the pathname format *PackageName::ElementName* (see Fig9 below).

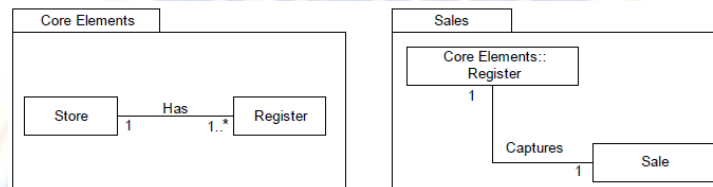


Fig9. A referenced class in a package.

### Package Dependencies

If a model element is in some way dependent on another, the dependency may be shown with a dependency relationship, depicted with an arrowed line.

A package dependency indicates that elements of the dependent package in some way know about or are coupled to elements in the target package.

For example, if a package references an element owned by another, a dependency exists. Thus, the *Sales* package has a dependency on the *Core Elements* package (see Fig10 below)

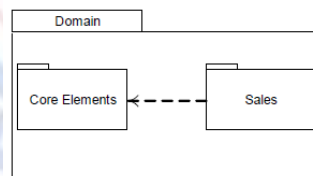


Fig10. A package dependency.

### Package Indication without Package Diagram

At times, it is inconvenient to draw a package diagram, but still desirable to indicate the package that the elements are a member of. In this situation, include a note (dog-eared note) on the diagram, as illustrated in Fig11 below.

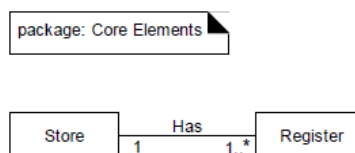


Fig11. Illustrating package ownership with a note.



### ***How to Partition the Domain Model***

How should the classes in a domain model be organized within packages? Apply the following general guidelines:

To partition the domain model into packages, place elements together that:

- are in the same subject area — closely related by concept or purpose
- are in a class hierarchy together
- participate in the same use cases
- are strongly associated

It is useful if all elements related to the domain model are rooted in a package called *Domain*, and all widely shared, common, core concepts are defined in a packaged named something like *Core Elements* or *Common Concepts*, in the absence of any other meaningful package within which to place them.

### ***POS Domain Model Packages***

Based on the above criteria, the package organization for the POS Domain Model is shown in Fig12 below.



Fig12. Domain concept packages.

### ***Core/Misc Package***

A Core/Misc package (see Fig13 below) is useful to own widely shared concepts or those without an obvious home. In later references, the package name will be abbreviated to *Core*.

There are no new concepts or associations particular to this iteration in this package.

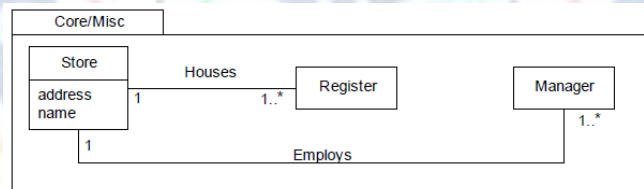


Fig13. Core package.

### ***Payments***

As in iteration 1, new associations are primarily motivated by a need-to-know criterion. For example, there is a need to remember the relationship between *CreditPayment* and *CreditCard*. In contrast, some associations are added more for comprehension, such as *DriversLicense Identifies Customer* (see Fig14 below).

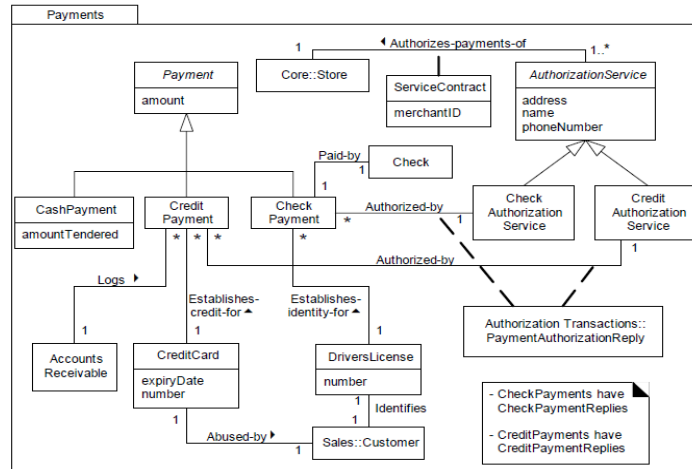


Fig14. Payments package.

**Products**

With the exception of composite aggregation, there are no new concepts or associations particular to this iteration (see Fig15 below).

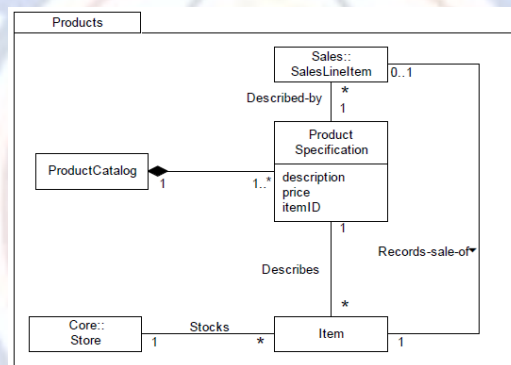


Fig15. Products package.

**Sales:**

With the exception of composite aggregation and derived attributes, there are no new concepts or associations particular to this iteration (see Fig16 below).

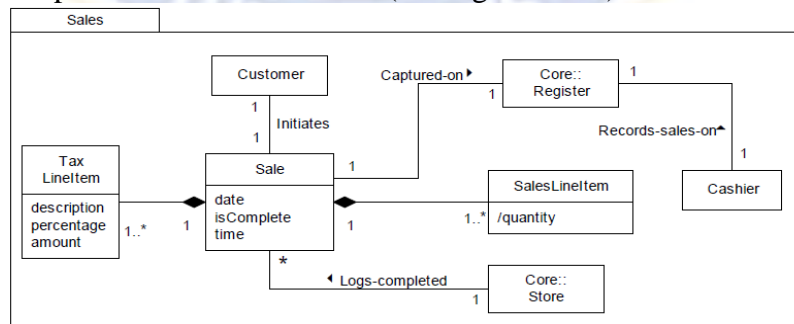


Fig16. Sales package.

**Authorization Transactions**

Although providing meaningful names for associations is recommended, in some circumstances it may not be compelling, especially if the purpose of the association is considered obvious to the audience. A case in point is the associations between payments and their transactions.

Their names have been left unspecified because we can assume the audience reading the class diagram in Fig17 given below will understand that the transactions are for the payment; adding the names merely makes the diagram more busy.

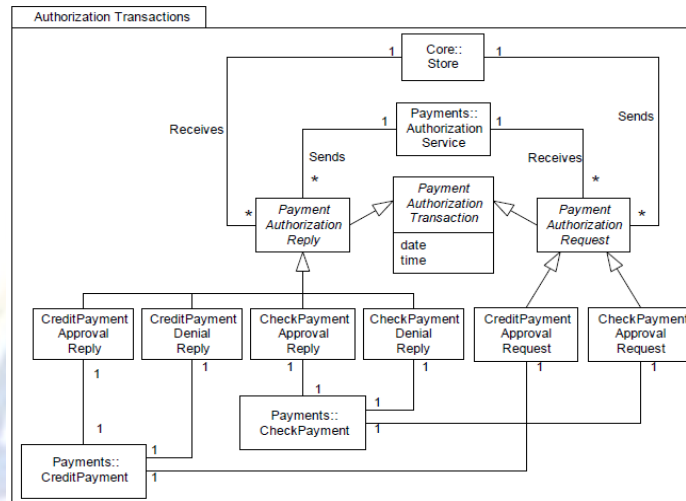


Fig17. Authorization transaction package.

**10. FINDING CONCEPTUAL CLASS HIERARCHIES**

**Explain about finding the conceptual class hierarchies.**

**Defining Conceptual Superclasses and Subclasses**

Since it is valuable to identify conceptual super- and subclasses, it is useful to clearly and precisely understand generalization, superclasses, and subclasses in terms of class definition and class sets.

**What is the relationship of a conceptual superclass to a subclass?**

A conceptual superclass definition is more general or encompassing than a subclass definition.

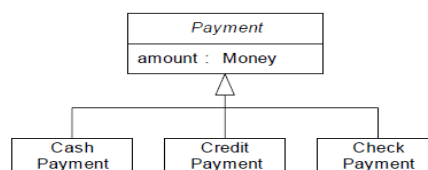


Fig18. Payment class hierarchy.

A *CreditPayment* is a transfer of money via a credit institution which needs to be authorized. My definition of *Payment* encompasses and is more general than my definition of *CreditPayment*.

### **Generalization and Class Sets**

Conceptual subclasses and superclasses are related in terms of set membership. All the members of a conceptual subclass set are members of their superclass set.

### **Conceptual Subclass Definition Conformance**

When a class hierarchy is created, statements about superclasses that apply to subclasses are made. For example, Fig19 below states that all *Payments* have an *amount* and are associated with a *Sale*.

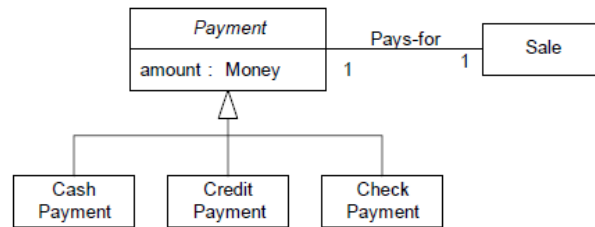


Fig19. Subclass conformance.

All *Payment* subclasses must conform to having an amount and paying for a *Sale*. In general, this rule of conformance to a superclass definition is the *100% Rule*:

#### **100% Rule**

*100% of the conceptual superclass's definition should be applicable to the subclass. The subclass must conform to 100% of the superclass's:*

- *attributes*
- *associations*

### **Conceptual Subclass Set Conformance**

A conceptual subclass should be a member of the set of the superclass. Thus, *CreditPayment* should be a member of the set of *Payments*.

Informally, this expresses the notion that the conceptual subclass *is a kind of* superclass. *CreditPayment is a kind of Payment*. More tersely, *is-a-kind-of* is called *is-a*.

This kind of conformance is the *Is-a Rule*:

#### **Is-a Rule**

*All the members of a subclass set must be members of their superclass set.*

### **What Is a Correct Conceptual Subclass?**

A potential subclass should conform to the:

- 100% Rule (definition conformance)
- Is-a Rule (set membership conformance)

### **Motivations to Partition a Conceptual Class into Subclasses**

The following are strong motivations to partition a class into subclass:

Create a conceptual subclass of a superclass when:

1. The subclass has additional attributes of interest.
2. The subclass has additional associations of interest.
3. The subclass concept is operated on, handled, reacted to, or manipulated differently than the superclass or other subclasses, in ways that are of interest.

4. The subclass concept represents an animate thing (for example, animal, robot) that behaves differently than the superclass or other subclasses, in ways that are of interest.

The following are motivations to generalize and define a superclass:

Create a conceptual superclass in a generalization relationship to subclasses when:

1. The potential conceptual subclasses represent variations of a similar concept.
2. The subclasses will conform to the 100% and Is-a rules.
3. All subclasses have the same attribute which can be factored out and expressed in the superclass.
4. All subclasses have the same association which can be factored out and related to the superclass.

The following sections illustrate these points.

### NextGen POS Conceptual Class Hierarchies

#### Payment Classes

Based on the above criteria for partitioning the *Payment* class, it is useful to create a class hierarchy of various kinds of payments. The justification for the superclass and subclasses is shown in Fig20 below.

#### Authorization Service Classes

Credit and check authorization services are variations on a similar concept, and have common attributes of interest. This leads to the class hierarchy in Fig21 below.

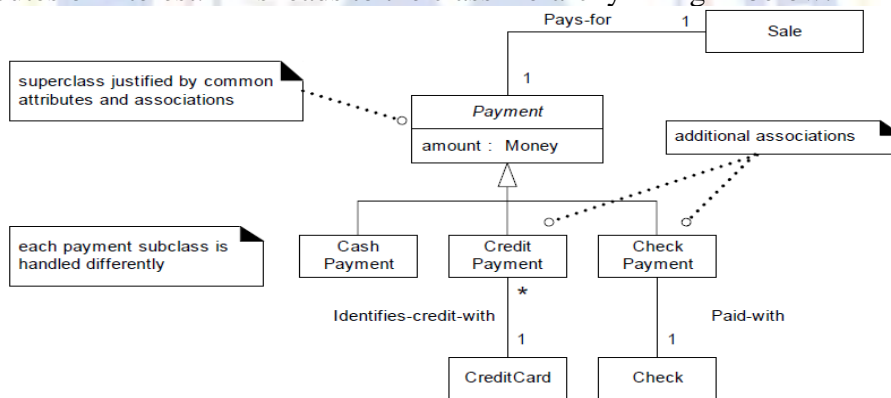


Fig20. Justifying Payment subclasses.

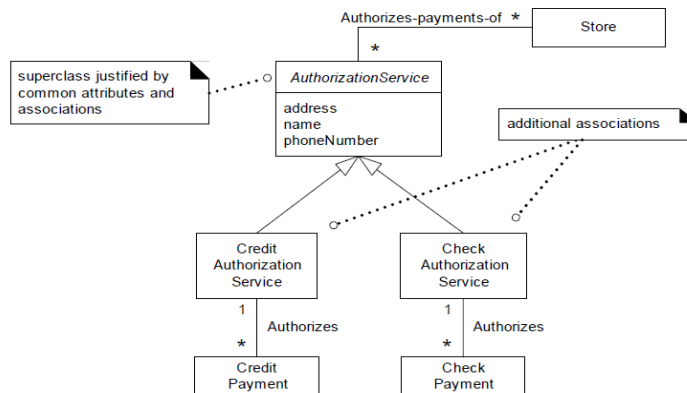


Fig21. Justifying the AuthorizationService hierarchy.



**Authorization Transaction Classes**

Modeling the various kinds of authorization service transactions (requests and replies) presents an interesting case. In general, transactions with external services are useful to show in a domain model because activities and processes tend to revolve around them. They are important concepts.

- Should the modeler illustrate *every* variation of an external service transaction?

It depends. As mentioned, domain models are not necessarily correct or wrong, but rather more or less useful. They are useful, because each transaction class is related to different concepts, processes, and business rules.

- A second interesting question is the degree of generalization that is useful to show in the model. For argument's sake, let us assume that every transaction has a date and time. These common attributes, plus the desire to create an ultimate generalization for this family of related concepts, justifies the creation of *PaymentAuthorizationTransaction*.

But is it useful to generalize a reply into a *CreditPaymentAuthorizationReply* and *CheckPaymentAuthorizationReply*, as shown in Fig21 below, or is it sufficient to show less generalization, as depicted in Fig22

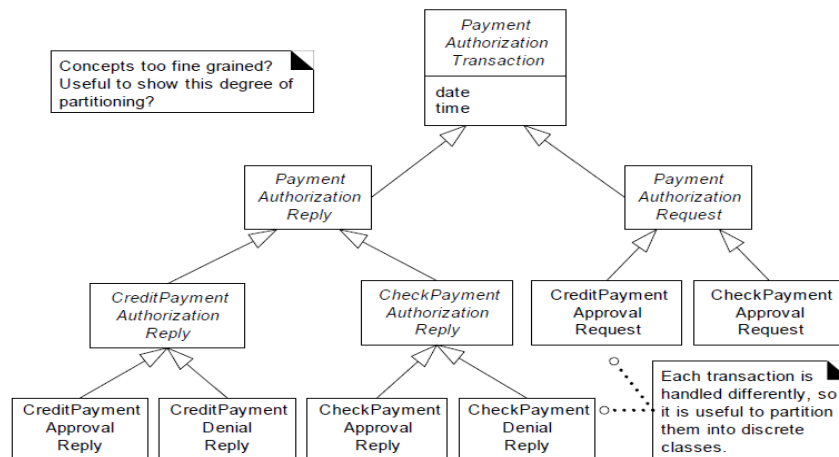


Fig21. One possible class hierarchy for external service transactions.

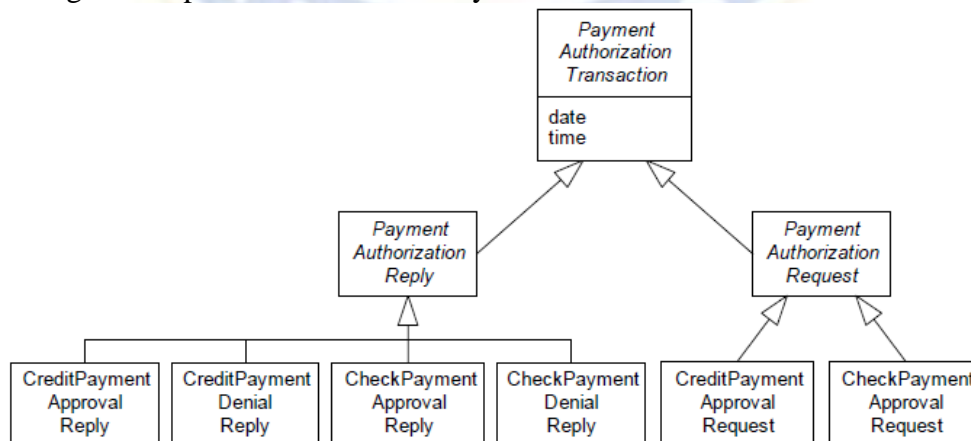


Fig22. An alternate transaction class hierarchy.

The class hierarchy shown in Figure22 is sufficiently useful in terms of generalization, because the additional generalizations do not add obvious value.

### Abstract Conceptual Classes

It is useful to identify abstract classes in the domain model because they constrain what classes it is possible to have concrete instances of, thus clarifying the rules of the problem domain.

If every member of a class C must also be a member of a subclass, then class C is called an abstract conceptual class.

### Abstract Class Notation in the UML

To review, the UML provides a notation to indicate abstract classes—the class name is italicized (see Fig23 below).

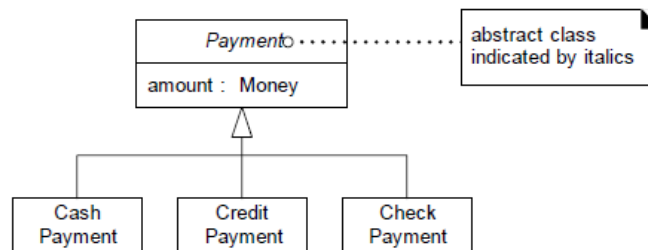


Fig23. Abstract class notation.

## 11. ASSOCIATIONS, AGGREGATION AND COMPOSITION:

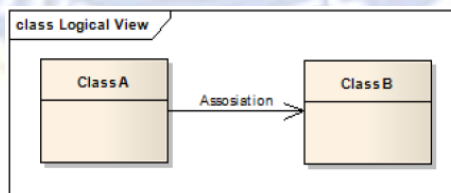
Explain in detail about the aggregation, association and composition. (Nov/Dec 2016, April/May 2017)

### UML CLASS DIAGRAM: ASSOCIATION, AGGREGATION AND COMPOSITION

The UML Class diagram is used to visually describe the problem domain in terms of types of object (classes) related to each other in different ways. There are three primary inter-object relationships: association, aggregation, and composition.

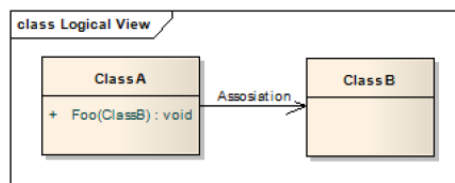
#### Association

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.



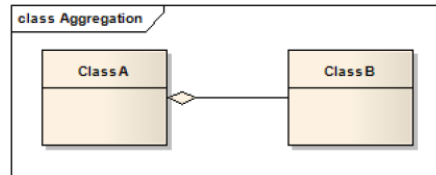
#### Weak Association

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.



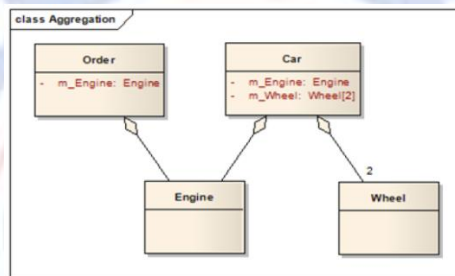
## AGGREGATION (SHARED ASSOCIATION)

In cases where there's a part-of relationship between ClassA (whole) and ClassB (part), we can be more specific and use the aggregation link instead of the association link, taking special notice that ClassB can also be aggregated by other classes in the application (therefore aggregation is also known as shared association).



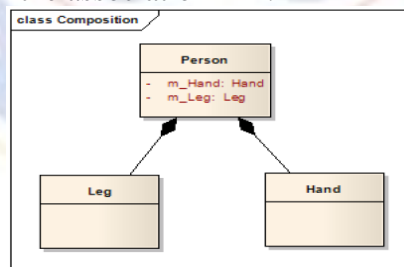
So basically, the aggregation link doesn't state in any way that ClassA owns ClassB or that there is a parent-child relationship (when parent deleted all its child's are being deleted as a result) between the two.

Actually, quite the opposite! The aggregation link usually used to stress the point that ClassA is not the exclusive container of ClassB, as in fact ClassB has another container.



## COMPOSITION (NOT-SHARED ASSOCIATION)

In cases where in addition to the part-of relationship between ClassA and ClassB - there's a strong life cycle dependency between the two, meaning that when ClassA is deleted then ClassB is also deleted as a result, we should be more specific and use the composition link instead of the aggregation link or the association link.



The composition link shows that a class (container, whole) has exclusive ownership over other class/s (parts), meaning that the container objects and its parts constitute a parent-child/s relationship.

Unlike association and aggregation, in the composition relationship, the composed class cannot appear as a return type or parameter type of the composite class, thus changes in the composed class cannot be propagated to the rest of the system. Consequently, usage of composition limits complexity growth as the system grows.

## CS6502- OBJECT ORIENTED ANALYSIS AND DESIGN

### UNIT – IV APPLYING DESIGN PATTERNS

System sequence diagrams - Relationship between sequence diagrams and use cases Logical architecture and UML package diagram – Logical architecture refinement - UML class diagrams -UML interaction diagrams - Applying GoF design patterns.

---

#### PRE-REQUISITE DISCUSSION:

- A Sequence diagram is an interaction diagram that shows how processes operate with one another and what is their order.
- It is a construct of a Message Sequence Chart.
- A sequence diagram shows object interactions arranged in time sequence. It depicts the objects and classes involved in the scenario and the sequence of messages exchanged between the objects needed to carry out the functionality of the scenario.
- Sequence diagrams are typically associated with use case realizations in the Logical View of the system under development. Sequence diagrams are sometimes called event diagrams or event scenarios.

#### 1. SYSTEM SEQUENCE DIAGRAMS

- The UML includes **sequence diagrams** as a notation that can illustrate actor interactions and the operations initiated by them.
- A **system sequence diagram** is a picture that shows, *for one particular scenario of a use case*, the events that external actors generate their order, and inter-system events. All systems are treated as a black box
- If the lifeline is that of an object, it demonstrates a role. Leaving the instance name blank can represent anonymous and unnamed instances.
- Messages, written with horizontal arrows with the message name written above them, display interaction. Solid arrow heads represent synchronous calls, open arrow heads represent asynchronous messages, and dashed lines represent reply messages.
- If a caller sends a synchronous message, it must wait until the message is done, such as invoking a subroutine. If a caller sends an asynchronous message, it can continue processing and doesn't have to wait for a response.
- Asynchronous calls are present in multithreaded applications and in message-oriented middleware. Activation boxes, or method-call boxes, are opaque rectangles drawn on top of lifelines to represent that processes are being performed in response to the message (Execution Specifications in UML).
- Objects calling methods on themselves use messages and add new activation boxes

on top of any others to indicate a further level of processing.

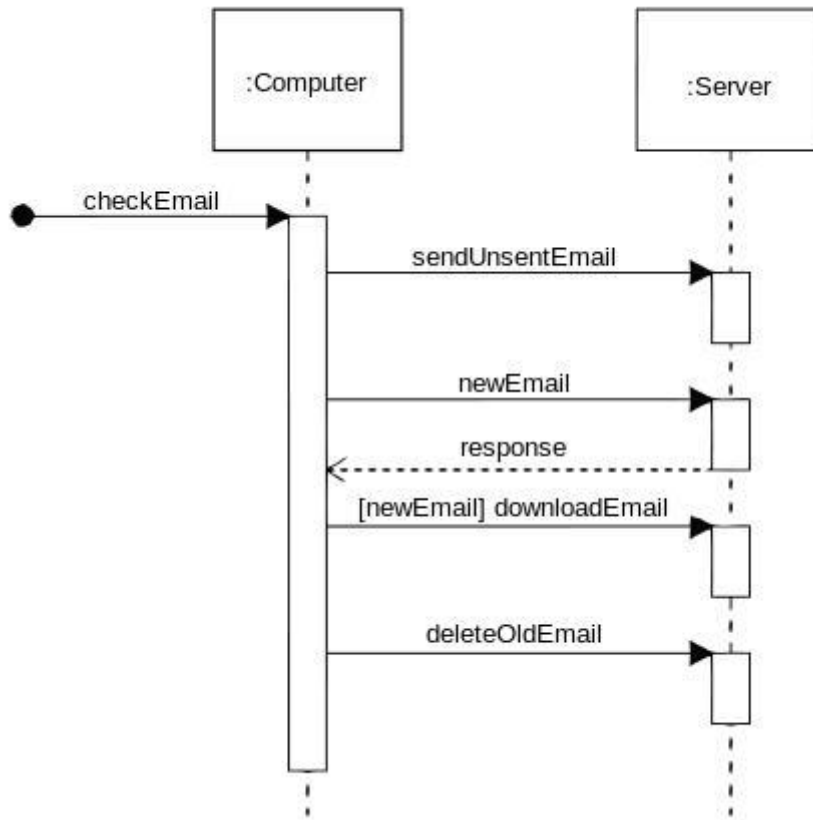
- When an object is destroyed (removed from memory), an X is drawn on top of the lifeline, and the dashed line ceases to be drawn below it (this is not the case in the first example though). It should be the result of a message, either from the object itself, or another.

- A message sent from outside the diagram can be represented by a message originating from a filled-in circle (found message in UML) or from a border of the sequence diagram (gate in UML).





○ UML has introduced significant improvements to the capabilities of sequence diagrams. Most of these improvements are based on the idea of interaction fragments which represent smaller pieces of an enclosing interaction. Multiple interaction fragments are combined to create a variety of combined fragments, which are then used to model interactions that include parallelism, conditional branches, and optional interactions.

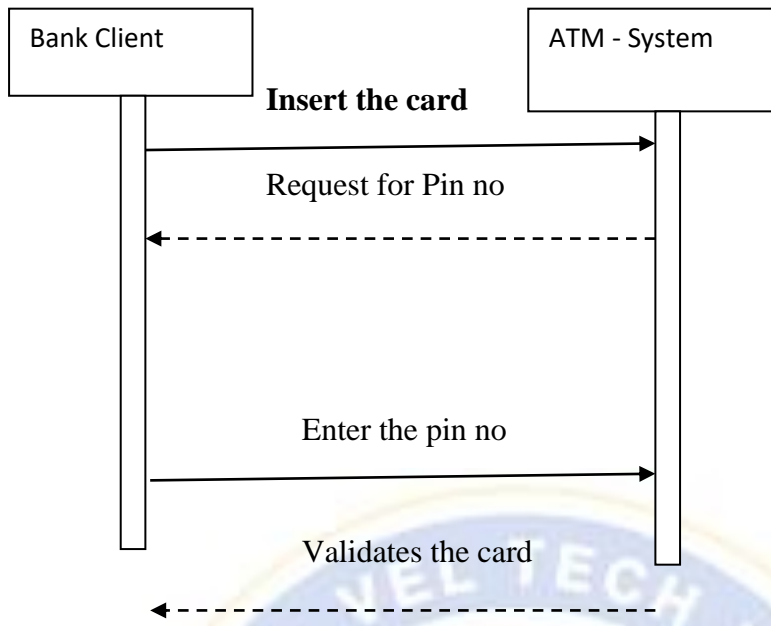


### Why Draw an SSD?

**System behavior** is a description of *what* a system does, without explaining how it does it. One part of that description is a system sequence diagram. Other parts include the use cases and system operation contracts

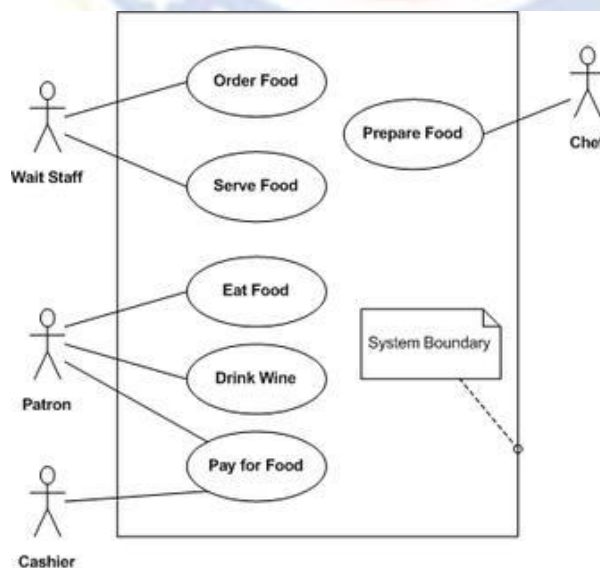
### Applying UML: Sequence Diagrams

**Sequence Diagram for ATM system**



**2. RELATIONSHIP BETWEEN SEQUENCE DIAGRAMS AND USE CASES**

**Illustrate with an example, the relationship between sequence diagrams and usecases.**  
**(May/June 2014)**

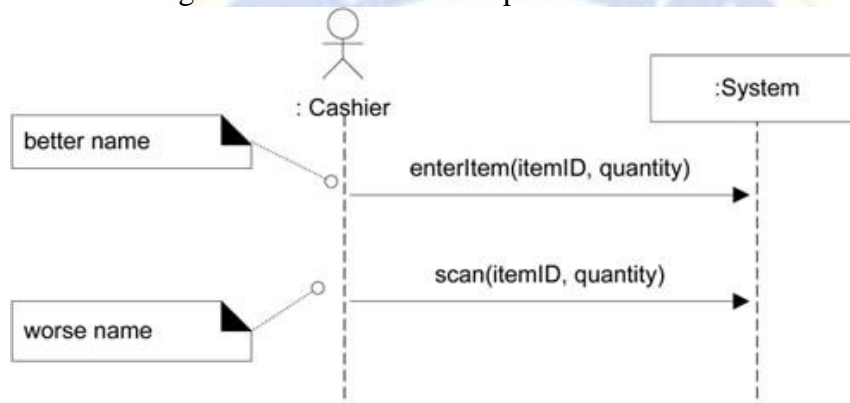


### How to Name System Events and Operations?

*Which is better, scan(itemID) or enterItem(itemID)?*

- System events should be expressed at the abstract level of intention rather than in terms of the physical input device.
- Thus "enterItem" is better than "scan" (that is, laser scan) because it captures the *intent* of the operation while remaining abstract and noncommittal with respect to design choices about what interface is used to capture the system event. It could be via laser scanner, keyboard, voice input, or anything.
- It also improves clarity to start the name of a system event with a verb (add..., enter..., end..., make...), as in Figure given below, since it emphasizes these are commands or requests.

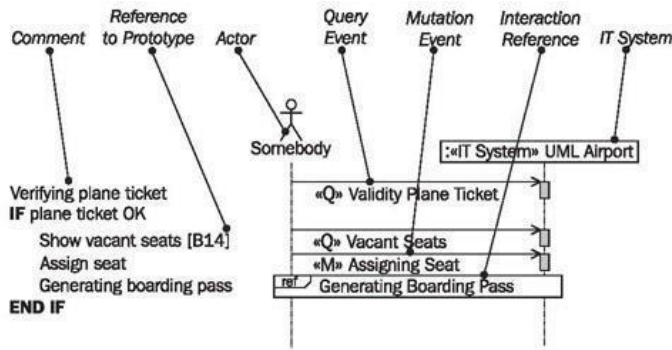
Figure. Choose event and operation names at an abstract level.



### Use Case Diagram:

- Realizing use cases by means of sequence diagrams is an important part of our analysis. It ensures that we have an accurate and complete class diagram.
- The sequence diagrams increase the completeness and understandability of our analysis model. Often, analysts use the sequence diagram to assign responsibilities to classes. The behavior is associated with the class the first time it is required, and then the behavior is reused for every other use case that requires the behavior.
- When assigning behaviors or responsibilities to a class while mapping a use case to the analysis model, you must take special care to assign the responsibility to the correct class. The responsibility or behavior belongs to the class if it is something you would do to the thing the class represents.
- For example, if our class represented a table and our application must keep track of the physical location of the table, we would expect to find the method MOVE in the class. We also expect the object to maintain all the information associated with an object of a given type.
- Therefore, the TABLE class should include the method WHERE LOCATED.
- This simple rule of thumb states that a class will include any methods needed to provide information about an object of the given class, will provide necessary methods to manipulate the objects of the given class, and will be responsible for creating and deleting instances of the class.
- Another guideline helpful in building an analysis model is to examine the model from the whole-part perspective.

- The start of the sequence diagram is normally the most difficult aspect. It is important to have access to the object on which you will call a method to start the sequence. It is often the case that you will have to return to the whole to navigate through the whole-part relationship to arrive at the class you will be working with.
- A simple example illustrates the whole-part relationship navigation. Suppose you were asked to read the first paragraph of three chapters of a book. First, you would need to know where to go to get the book. We might state that all books we are referring to are available at the Fourth St. library.



### 3. LOGICAL ARCHITECTURE AND UML PACKAGE DIAGRAM

#### Explain the Logical architecture and UML package diagram. (Nov/Dec 2011)

##### What is the Logical Architecture? And Layers?

The **logical architecture** is the large-scale organization of the software classes into packages (or namespaces), subsystems, and layers. It's called the *logical* architecture because there's no decision about how these elements are deployed across different operating system processes or across physical computers in a network (these latter decisions are part of the **deployment architecture**).

A **layer** is a very coarse-grained grouping of classes, packages, or subsystems that has cohesive responsibility for a major aspect of the system. Also, layers are organized such that "higher" layers (such as the UI layer) call upon services of "lower" layers, but not normally vice versa.

Typically layers in an OO system include:

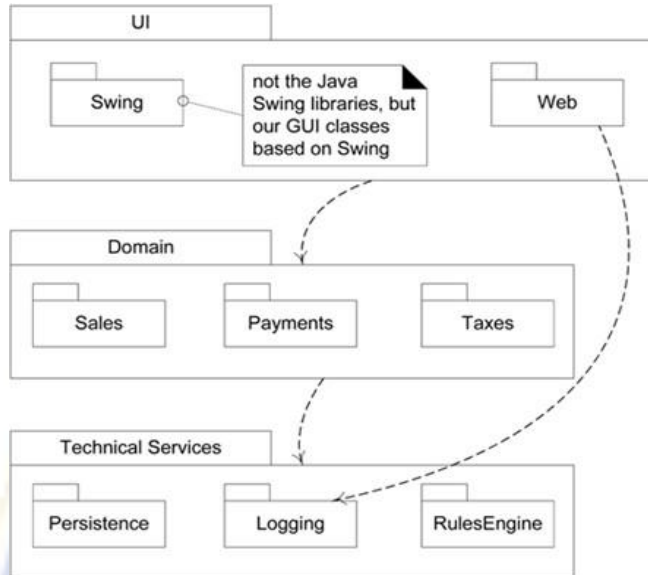
- **User Interface**
- **Application Logic and Domain Objects** software objects representing domain concepts (for example, a software class *Sale*) that fulfill application requirements, such as calculating a sale total.
- **Technical Services** general purpose objects and subsystems that provide supporting technical services, such as interfacing with a database or error logging. These services are usually application-independent and reusable across several systems.

In a **strict layered architecture**, a layer only calls upon the services of the layer directly below it.

This design is common in network protocol stacks, but not in information systems, which usually have a **relaxed layered architecture**, in which a higher layer calls upon several lower layers. For example, the UI layer may call upon its directly subordinate application logic layer, and also upon elements of a lower technical service layer, for logging and so forth.

A logical architecture doesn't have to be organized in layers. But it's *very* common, and hence, introduced at this time.

Fig1. Layers shown with UML package diagram notation.



**SAMPLE UP ARTIFACT RELATIONSHIPS**

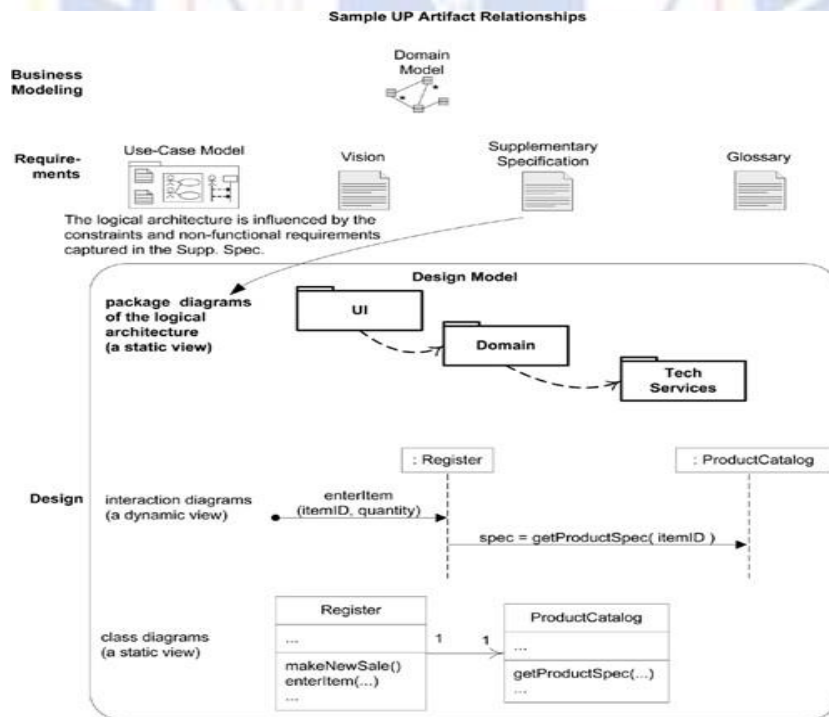




Fig2. Sample UP artifact influence.

### What are the Layers Focussed?

OOAD focuses on :

1. Core application logic or domain layer.
2. UI layer
3. Other layers

### What is Software Architecture?

Architecture is the set of significant decisions about the organization of a software system, the selection of the structural elements and their interfaces by which the system is composed, together with their behavior as specified in the collaborations among those elements

### Applying UML: Package Diagrams

1. UML package diagrams are often used to illustrate the logical architecture of a system the layers, subsystems, packages.
2. A layer can be modeled as a UML package; for example, the UI layer modeled as a package named UI.
3. A UML package diagram provides a way to group elements. A UML package can group anything: classes, other packages, use cases.
4. The package name may be placed on the tab if the package shows inner members, or on the main folder, if not.
5. The UML **dependency line (dashed arrow line) with the arrow pointing towards the depended-on .package can be used to show coupling.**
6. A UML package represents a **namespace.**
7. The UML provides alternate notations to illustrate outer and inner nested packages. Sometimes it is awkward to draw an outer package box around inner packages.

### What is Mode-View-Separation principle? Explain the motivation for model-view separation. (May/June 2016)

#### Guideline: The Model-View Separation Principle

In this context, *model* is a synonym for the domain layer of objects. *View* is a synonym for UI objects, such as windows, Web pages, applets, and reports.

The *Model - View Separation* principle<sup>2</sup> states that model (domain) objects should not have *direct* knowledge of view (UI) objects, at least as view objects. So, for example, a *Register* or *Sale* object should not directly send a message to a GUI window object *ProcessSaleFrame*, asking it to display something, change color, close, and so forth.

A legitimate relaxation of this principle is the Observer pattern, where the domain objects send messages to UI objects viewed only in terms of an interface such as *PropertyListener* (a common Java interface for this situation).

Then, the domain object doesn't know that the UI object is a UI object - it doesn't know its concrete window class. It only knows the object as something that implements the *PropertyListener* interface.

A further part of this principle is that the domain classes encapsulate the information and behavior related to application logic. The window classes are relatively thin; they are responsible for input and output, and catching GUI events, but *do not* maintain application data or directly provide application logic.

For example, a Java *JFrame* window should *not* have a method that does a tax calculation. A Web JSP page should *not* contain logic to calculate the tax. These UI elements should delegate to non - UI elements for such responsibilities.

The *motivation for Model - View Separation* includes:

- To support cohesive model definitions that focus on the domain processes, rather than on user interfaces.
- To allow separate development of the model and user interface layers.
- To minimize the impact of requirements changes in the interface upon the domain layer.
- To allow new views to be easily connected to an existing domain layer, without affecting the domain layer.
- To allow multiple simultaneous views on the same model object, such as both a tabular and business chart view of sales information.
- To allow execution of the model layer independent of the user interface layer, such as in a message - processing or batch - mode system.
- To allow easy porting of the model layer to another user interface framework.

### **What's the Connection Between SSDs, System Operations, and Layers?**

- During analysis work, we sketched some SSDs for use case scenarios. We identified input events from external actors into the system, calling upon system operations such as *makeNewSale* and *enterItem*.
- The SSDs illustrate these system operations, but hide the specific UI objects. Nevertheless, normally it will be objects in the UI layer of the system that capture these system operation requests, usually with a rich client GUI or Web page.
- In a well - designed layered architecture that supports high cohesion and a separation of concerns, the UI layer objects will then forward - or delegate - the request from the UI layer onto the domain layer for handling.
- Now, here's the key point:
- The messages sent from the UI layer to the domain layer will be the messages illustrated on the SSDs, such as *enterItem*.
- For example, in Java Swing, perhaps a GUI window class called *ProcessSale - Frame* in the UI layer will pick up the mouse and keyboard events requesting to enter an item, and then the *ProcessSaleFrame* object will send an *enterItem* message on to a software object in the domain layer, such as *Register*, to perform the application logic.
- System operations in the SSDs and in terms of layers



#### **4. LOGICAL ARCHITECTURE REFINEMENT**

**Explain Logical architecture refinement. (May/June 2015, Nov/Dec 2015)**

**NextGen Logical Architecture**

Fig3 given below illustrates a partial logical layered architecture for this iteration of NextGen application.

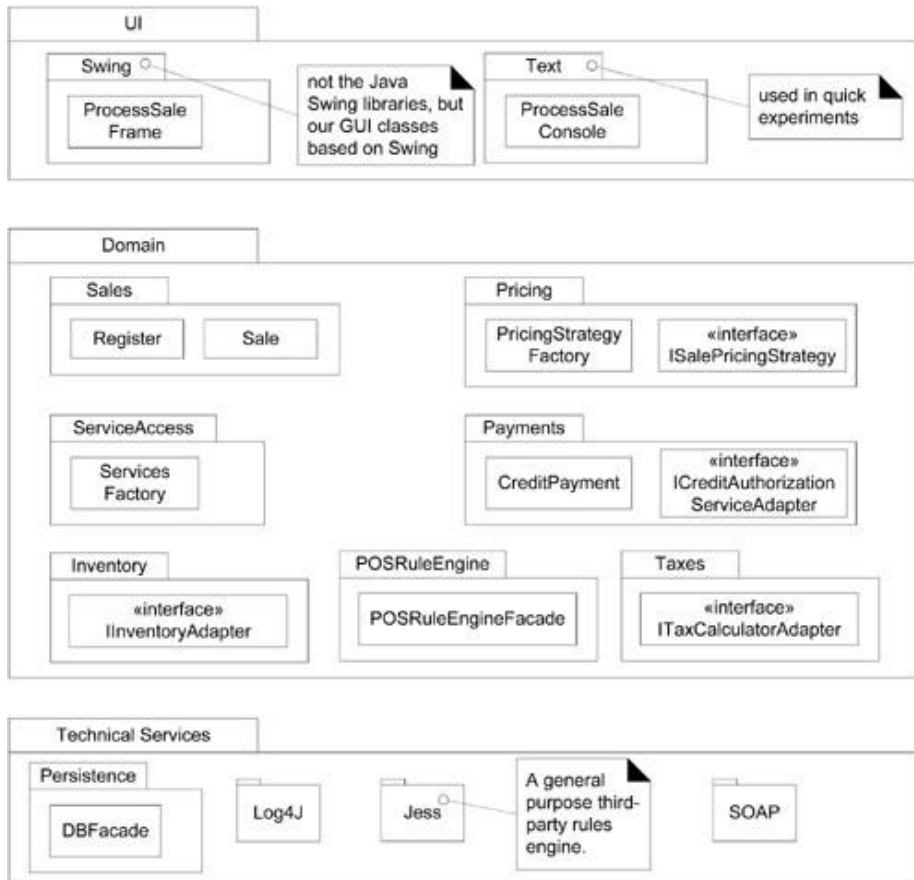
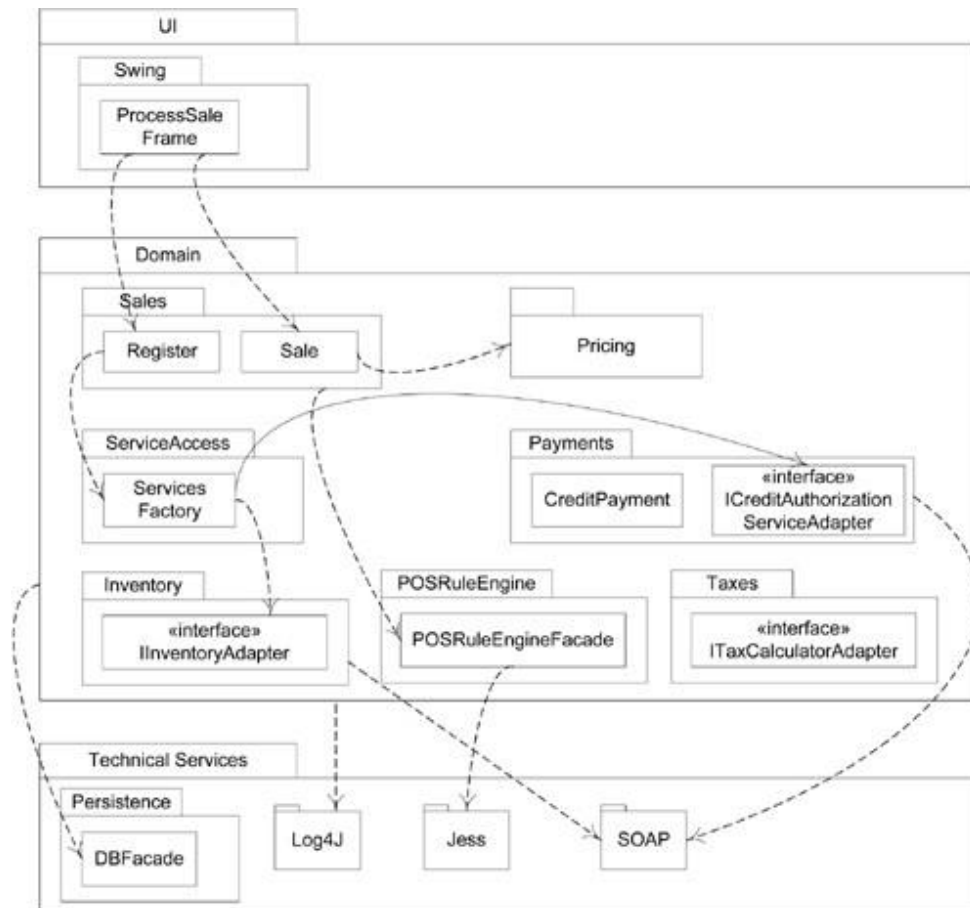


Fig3. Partial logical view of layers in the NextGen application

The 3 layers of partial logical layered architecture are:

- i) User Interface Layer
- ii) Domain Layer
- iii) Technical Services Layer



The above diagram illustrates the coupling between layers and packages.

- UI layer allows the user to manipulate a system and to indicate the effects of user's manipulation. Java Swing can be used to design UI.
- Domain layer is responsible for representing concepts of the business, information about the business situation and business rules.
- Technical Services Layer is used to depict high level or low level technical services such as persistence.

### 1. Inter-Layer and Inter-Package Coupling

#### Applying UML:

- Dependency lines can be used to communicate coupling between packages or types in packages. Plain dependency lines are excellent when the communicator does not care to be more specific on the exact dependency (attribute visibility, subclassing)but just wants to highlight general dependencies.
- Many elements of the package may share the dependency.
  - a) From Process Sale Frame to Register
  - b) From Process Sale Frame to sale.
  - c) From sale to PosRule Engine Façade.
  - d) From Inventory to SOAP.



## 2. Partial Package Coupling

It illustrates the coupling between packages.

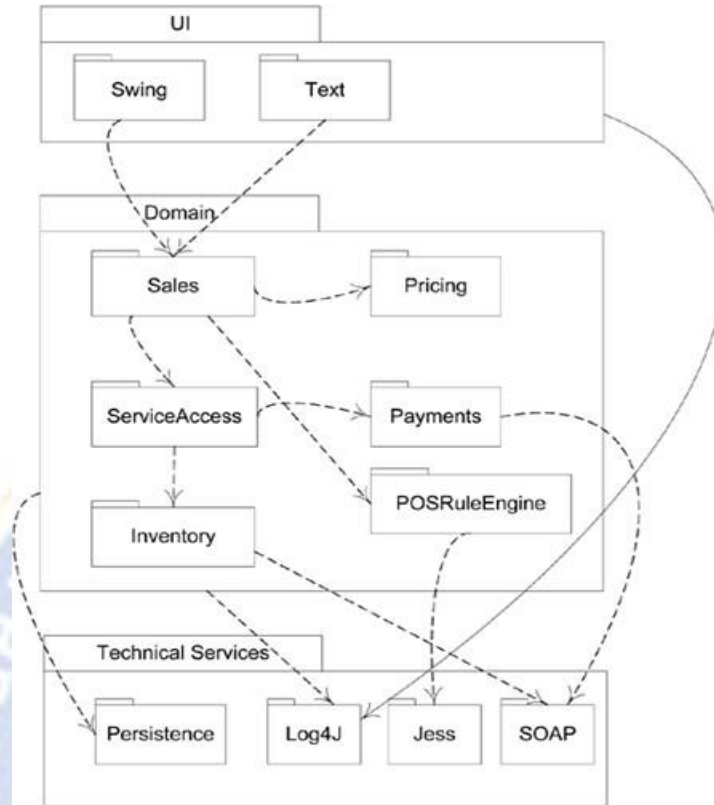


Fig4. Partial package coupling.

## 3. Inter-Layer and Inter-Package Interaction Scenarios

- Package diagrams show static information.
- To help understand the *dynamics* in the NextGen logical architecture, it's also useful to include a diagram of how objects across the layers connect and communicate. Thus, an interaction diagram is helpful.
- Logical view of the architecture focuses on the collaborations as they cross layer and package boundaries.

- A set of interaction diagrams illustrate **architecturally significant scenarios** that depicts the large-scale or big ideas in the design.

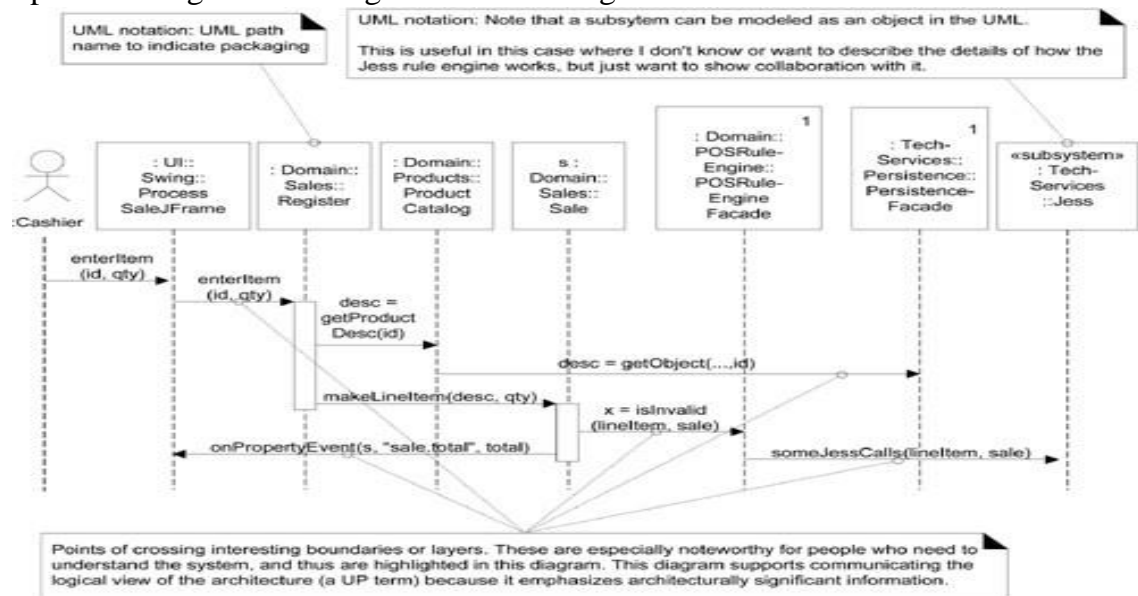


Fig5. An architecturally significant interaction diagram that emphasizes cross-boundary connections.

### Applying UML:

- The package of a type can optionally be shown by qualifying the type with the UML **path name** expression `<PackageName>::<TypeName>`. For example, `Domain::Sales::Register`. This can be exploited to highlight to the reader the inter-package and inter-layer connections in the interaction diagram.
- use of the `«subsystem»` stereotype. In the UML, a subsystem is a discrete entity that has behavior and interfaces. A subsystem can be modeled as a special kind of package, or as shown here as an object, which is useful when one wants to show inter-subsystem (or system) collaborations. In the UML, the entire system is also a "subsystem" (the root one), and thus can also be shown as an object in interaction diagrams (such as an SSD).
- use of '1' in the top right corner to indicate a singleton, and suggest access using the GoF Singleton pattern.

### Collaborations with the Layers Pattern

Two design decisions at an architectural level are:

- What are the big parts?
- How are they connected?

### Simple Packages versus Subsystems

- Packages groups the factory and strategies used.
- Packages are used to represent the different layers of source code.
- Example: Pricing, Payroll, Foundation Packages such as `java.util`
- Subsystems are a type of stereotyped component that represent independent, behavioral units in a system.
- Example: Persistence, POS Rule Engine, Jess.
- **Microarchitectural design patterns used for connection between layers and packages**

#### 1.Facade

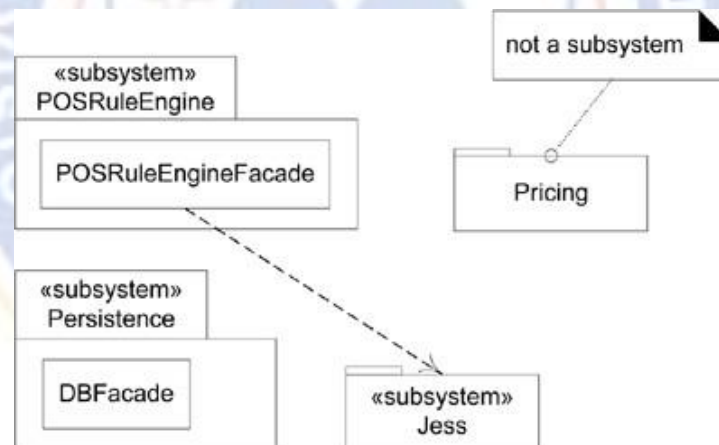
#### 2. Controller

### 3.Observer

#### Facade:

- For packages that represent subsystems, the most common pattern of access is Facade, a GoF design pattern.
- A public facade object defines the services for the subsystem, and clients collaborate with the facade, not internal subsystem components.
- This is true of the *POSRuleEngineFacade* and the *PersistenceFacade* for access to the rules engine and persistence subsystem.

Fig6. Subsystem stereotypes.



#### Session Facades and the Application Layer

- When an application has many system operations and supports many use cases, it is common to have more than one object mediating between the UI and Domain layers.
- In the current version of the NextGen system, there is a simple design of a single *Register* object acting as the facade onto the Domain layer (by virtue of the GRASP controller pattern).
- *Session Facades* are ones where each session instance represents a session with one client.

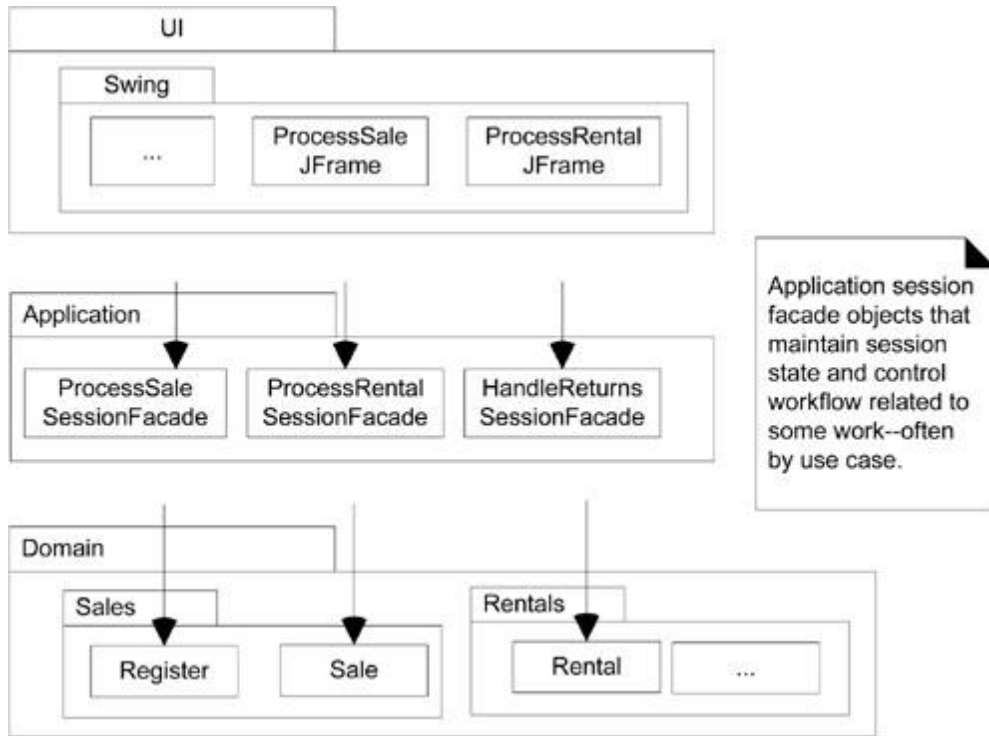


Fig7. Session facades and an Application Layer.

### Controller

The GRASP Controller pattern describes common choices in client-side handlers for system operation requests emitting from the UI layer.

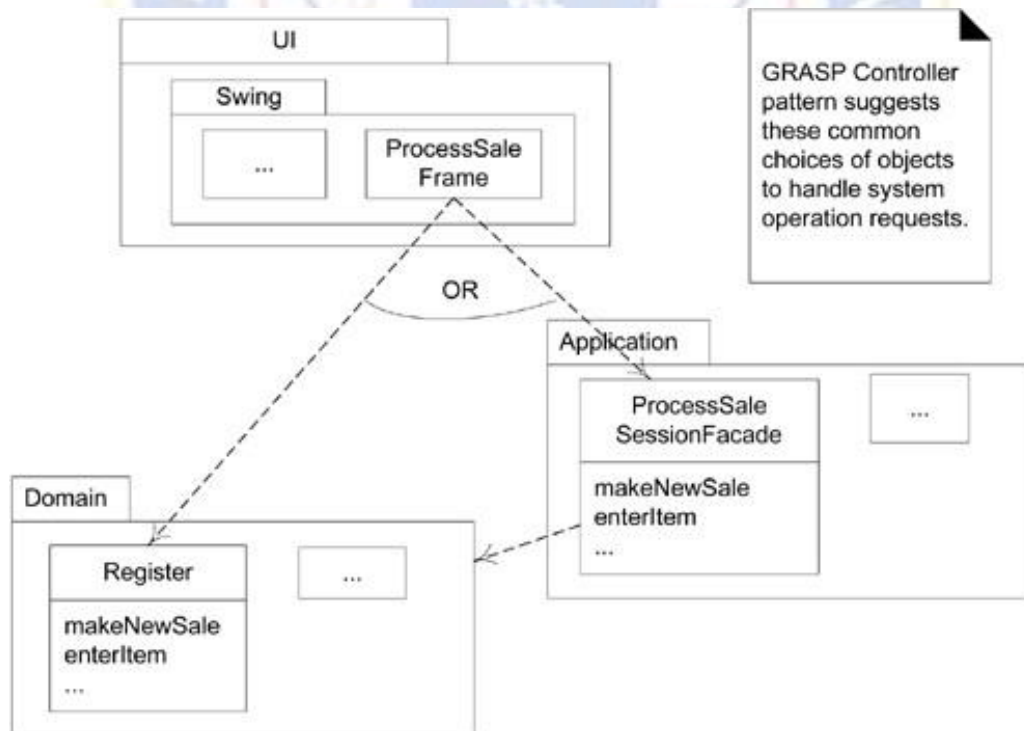
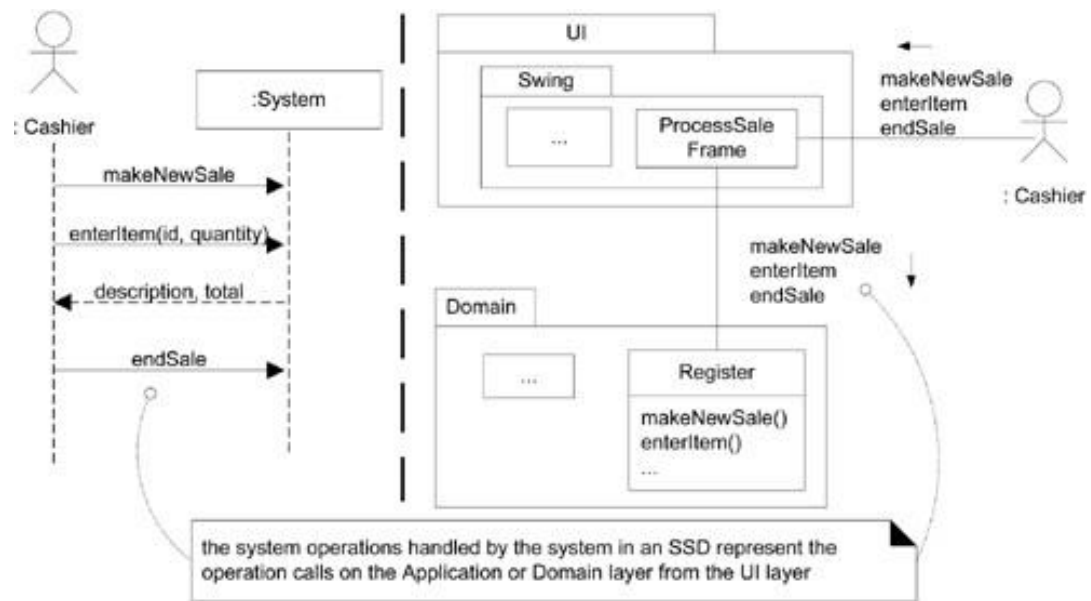


Fig8. The Controller choices.

### System Operations and Layers

The SSDs illustrate the system operations, hiding UI objects from the diagram. The system operations being invoked on the system are requests being generated by an actor via the UI layer, onto the Application or Domain layer.

Fig9. System operations in the SSDs and in terms of layers.

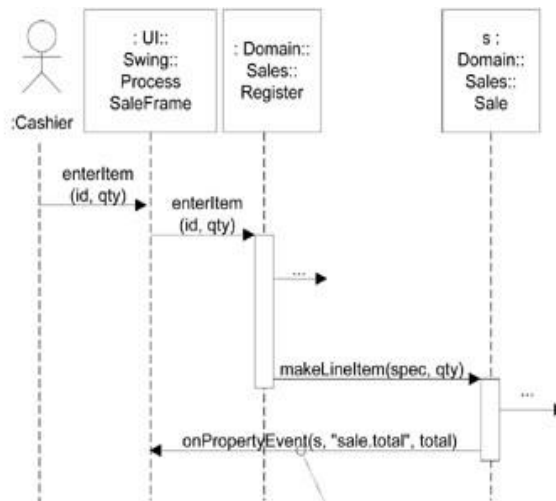


### Upward Collaboration with Observer

- The Facade pattern is commonly used for "downward" collaboration from a higher to lower layer, or for access to services in another subsystem of the same layer.
- When the lower Application or Domain layer needs to communicate upward with the UI layer, it is usually via the Observer pattern.
- That is, UI objects in the higher UI layer implement an interface such as *PropertyListener* or *AlarmListener*, and are subscribers or listeners to events (such as property or alarm events) coming from objects in the lower layers.
- The lower layer objects are directly sending messages to the upper layer UI objects, but the coupling is only to the objects viewed as things that implement an interface, such as *PropertyListener*, not viewed as specific GUI windows.

Fig10. Observer for "upward" communication to the UI layer.





Collaboration from the lower layers to the UI layer is usually via the Observer (Publish-Subscribe) pattern. The Sale object has registered subscribers that are PropertyListeners. One happens to be a Swing GUI JFrame, but the Sale does not know this object as a GUI JFrame, but only as a PropertyListener.

### Relaxed Layered Coupling

- The layers in most layered architectures are *not* coupled in the same limited sense as a network protocol based on the OSI 7-Layer Model.
- In the protocol model, there is strict restriction that elements of layer N only access the services of the immediate lower layer N-1.
- The standard is a "relaxed layered" or "transparent layered" architecture in which elements of a layer collaborate with or are coupled to several other layers.

### Coupling between layers:

- All higher layers have dependencies on the Technical Services and Foundations layer.
- For example, in Java all layers depend on *java.util* package elements.
- It is primarily the Domain layer that has dependency on the Business Infrastructure layer.
  - The UI layer makes calls on the Application layer, which makes service calls on the Domain layer; the UI layer does not call on the Domain, unless there is no Application layer.
  - If it is a single-process "desktop" application, software objects in the Domain layer are directly visible to, or passed between, UI, Application, and to a lesser extent, Technical Services.

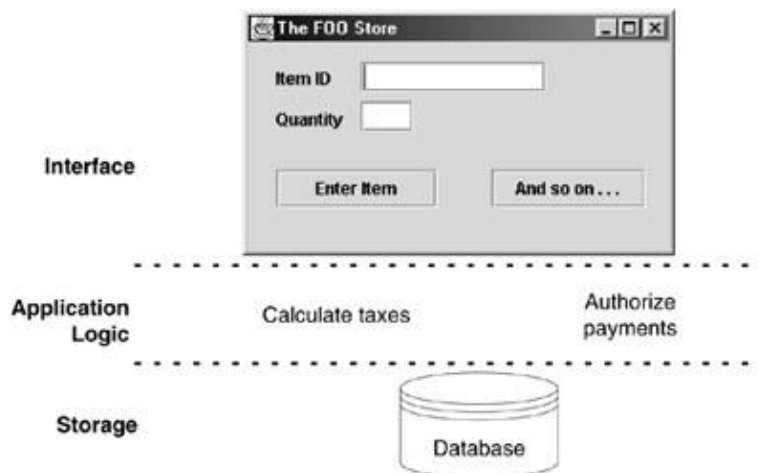
### Virtual Machines and Operating Systems

- In Operating systems, the "lower" layers encapsulated access to the physical resources and provided process and I/O services, and higher layers called on these services.
- These included Multics and the system.
- The idea of a virtual machine (VM) with a bytecode universal machine language is that applications could be written at higher layers in the architecture (and executed without recompilation across different platforms), on top of the virtual machine layer, which in turn would sit on top of the operating system and machine resources..

## Information Systems: The Classic Three-Tier Architecture

Description of a layered architecture for information systems that included a user interface and persistent storage of data was known as a **three-tier architecture**.

Fig11. Classic view of a three-tier architecture.

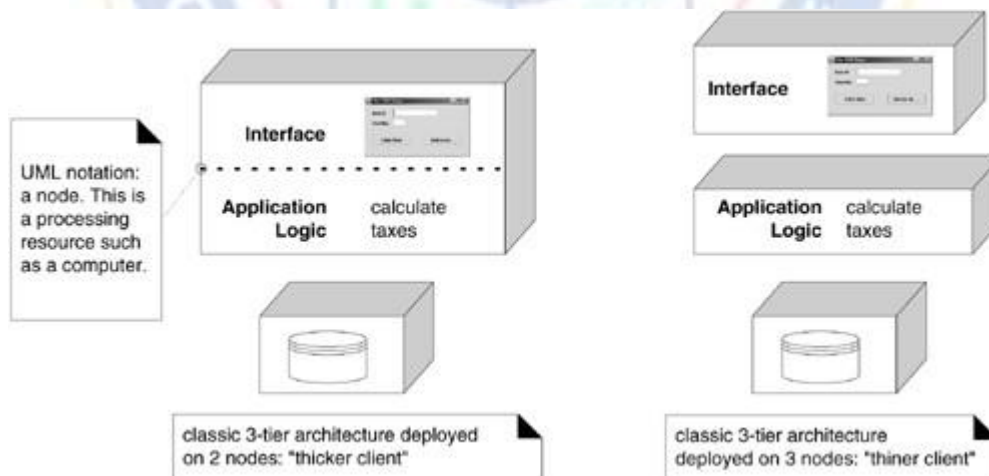


A classic description of the vertical tiers in a three-tier architecture is:

1. **Interface** windows, reports, and so on.
2. **Application Logic** tasks and rules that govern the process.
3. **Storage** persistent storage mechanism.

The singular quality of three-tier architecture is the separation of the application logic into a distinct logical middle tier of software. The interface tier is relatively free of application processing; windows or Web pages forward task requests to the middle tier. The middle tier communicates with the back-end storage layer.

Fig12. A three-tier logical division deployed in two physical architectures.



The three-tier architecture was contrasted by the Gartner Group with a **two-tier** design, in which, for example, application logic is placed within window definitions, which read and write directly to a database; there is no middle tier that separates out the application logic. Two-tier client-server architectures became especially popular with the rise of tools such as Visual Basic and PowerBuilder.

Two-tier designs have (in some cases) the advantage of initial quick development, but can suffer the complaints covered in the *Problems* section. Nevertheless, there are applications that are primarily simple CRUD (create, retrieve, update, delete) data intensive systems, for which this is a suitable choice.

## Related Patterns

- Indirection layers can add a level of indirection to lower-level services.
- Protected Variation layers can protect against the impact of varying implementations.
- Low Coupling and High Cohesion layers strongly support these goals.
- Its application specifically to object-oriented information systems

## **5. UML CLASS DIAGRAM**

The class diagram is the main building block of object oriented modelling. It is used both for general conceptual modelling of the systematics of the application, and for detailed modelling translating the models into programming code. Class diagrams can also be used for data modeling.

The classes in a class diagram represent both the main objects, interactions in the application and the classes to be programmed.

The top part contains the name of the class. It is printed in bold and centered, and the first letter is capitalized.

The middle part contains the attributes of the class. They are left-aligned and the first letter is lowercase.

The bottom part contains the methods the class can execute. They are also left-aligned and the first letter is lowercase.

In the design of a system, a number of classes are identified and grouped together in a class diagram which helps to determine the static relations between those objects. With detailed modelling, the classes of the conceptual design are often split into a number of subclasses.

- Links
- A Link is the basic relationship among objects.
- Association

An association represents a family of links. A binary association (with two ends) is normally represented as a line. An association can link any number of classes. An association with three links is called a ternary association. An association can be named, and the ends of an association can be adorned with role names, ownership indicators, multiplicity, visibility, and other properties.

There are four different types of association: bi-directional, uni-directional, Aggregation (includes Composition aggregation) and Reflexive. Bi-directional and uni-directional associations are the most common ones. For instance, a flight class is associated with a plane class bi-directionally. Association represents the static relationship shared among the objects of two classes.

### **Aggregation**

Aggregation is a variant of the "has a" association relationship; aggregation is more specific than association. It is an association that represents a part-whole or part-of relationship. As shown in the image, a Professor 'has a' class to teach. As a type of association, an aggregation can be named and have the same adornments that an association can. However, an aggregation may not involve more than two classes; it must be a binary association.

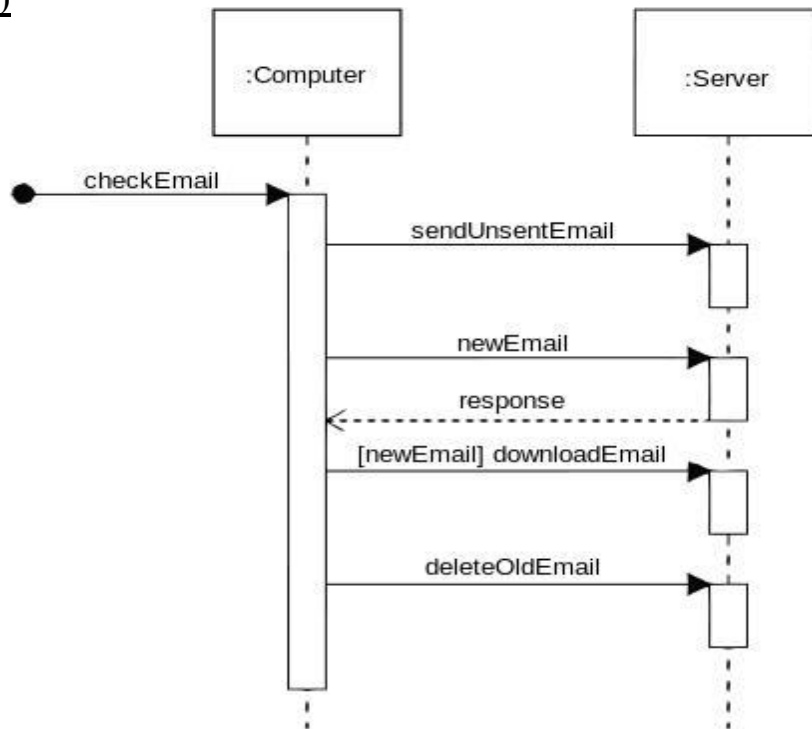
Aggregation can occur when a class is a collection or container of other classes, but the contained classes do not have a strong lifecycle dependency on the container. The contents of the container are not automatically destroyed when the container is.

In UML, it is graphically represented as a hollow diamond shape on the containing class with a single line that connects it to the contained class. The aggregate is semantically an extended object that is treated as a unit in many operations, although physically it is made of several lesser objects.



## 6. UML INTERACTION DIAGRAMS

**What do you mean by interaction diagrams? Explain them with a suitable example. (Nov/Dec 2011, 2015, May 2015)**



Interaction Overview Diagram is one of the thirteen types of diagrams of the Unified Modeling Language (UML), which can picture a control flow with nodes that can contain interaction diagrams.

The interaction overview diagram is similar to the activity diagram, in that both visualize a sequence of activities.

The difference is that, for an interaction overview, each individual activity is pictured as a frame which can contain a nested interaction diagrams. This makes the interaction overview diagram useful to "deconstruct a complex scenario that would otherwise require multiple if-then-else paths to be illustrated as a single sequence diagram".

The other notation elements for interaction overview diagrams are the same as for activity diagrams.



These include initial, final, decision, merge, fork and join nodes. The two new elements in the interaction overview diagrams are the "interaction occurrences" and "interaction elements."

## 7. APPLYING GOF DESIGN PATTERNS

### Designing the Use-Case realizations with GoF (Gang-of-Four) Design Patterns. (April/May 2011, May/June 2016)

#### The Gang-of-Four Design Patterns

GoF design patterns were first described in *Design Patterns*, a seminal and extremely popular work that presents patterns useful during object design.

#### Adapter (GoF)

**Name:** Adapter

**Problem:** How to resolve incompatible interfaces, or provide a stable interface to similar components with different interfaces?

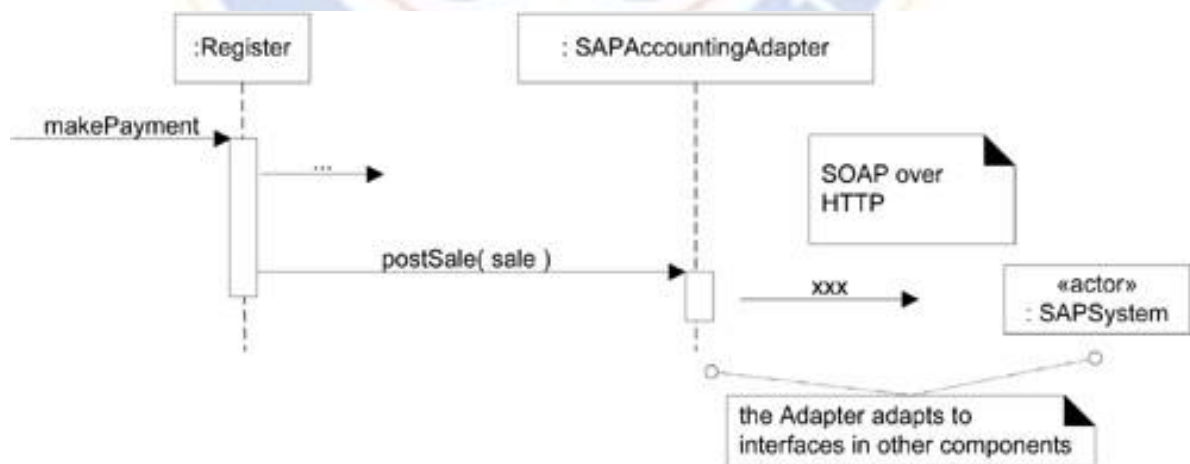
**Solution:** Convert the original interface of a component into another interface, through an intermediate adapter object.  
(advice)

**To review:** The NextGen POS system needs to support several kinds of external third-party services, including tax calculators, credit authorization services, inventory systems, and accounting systems, among others. Each has a different API, which can't be changed.

A solution is to add a level of indirection with objects that adapt the varying external interfaces to a consistent interface used within the application.

A particular adapter instance will be instantiated for the chosen external service, such as SAP for accounting, and will adapt the *postSale* request to the external interface, such as a SOAP XML interface over HTTPS for an intranet Web service offered by SAP.

Fig13. Using an Adapter.



#### Some GRASP Principles as a Generalization of Other Patterns

The previous use of the Adapter pattern can be viewed as a specialization of some GRASP building blocks:

Adapter supports *Protected Variations* with respect to changing external interfaces or third-party packages through the use of an *Indirection* object that applies interfaces and *Polymorphism*.

## What's the Problem? Pattern Overload!

The *Pattern Almanac 2000* lists around 500 design patterns. And many hundreds more have been published since then. The curious developer has no time to actually program given this reading list!

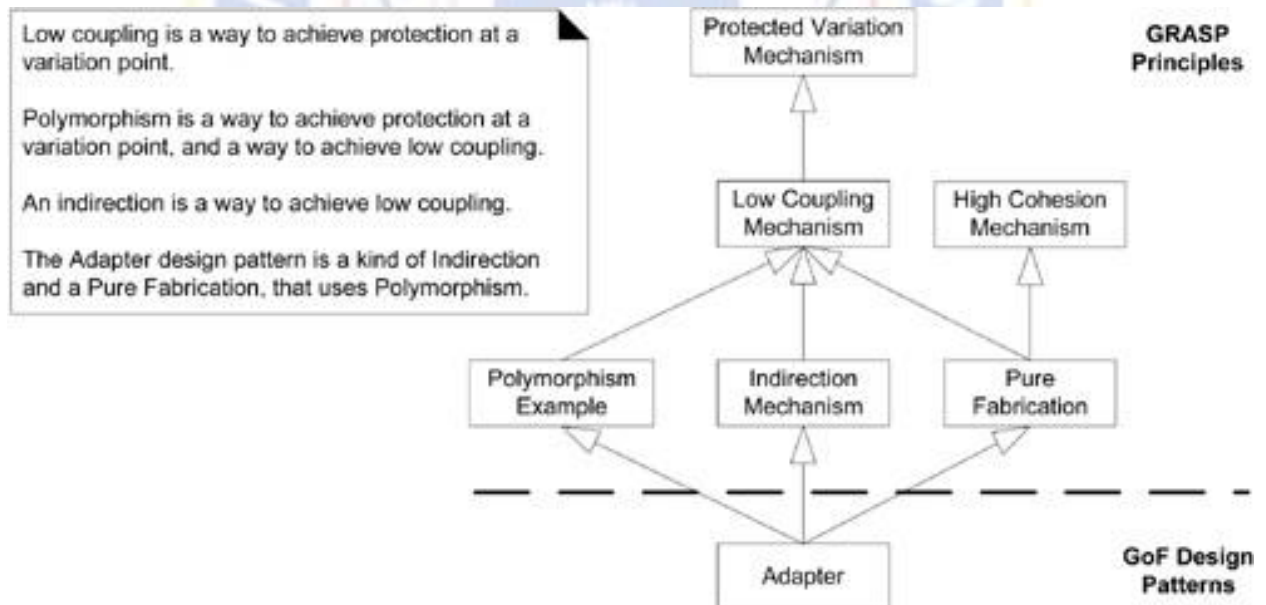
## A Solution: See the Underlying Principles

Yes, it's important for an experienced designer to know in detail and by memory 50+ of the most important design patterns, but few of us can learn or remember 1,000 patterns, or even start to organize that pattern plethora into a useful taxonomy.

But there's good news: Most design patterns can be seen as specializations of a few basic GRASP principles. Although it is indeed helpful to study detailed design patterns to accelerate learning, it is even more helpful to see their underlying basic themes (Protected Variations, Polymorphism, Indirection, ...) to help us to cut through the myriad details and see the essential "alphabet" of design techniques being applied.

## Example: Adapter and GRASP

Figure given below illustrates my point that detailed design patterns can be analyzed in terms of the basic underlying "alphabet" of GRASP principles. UML generalization relationships are used to suggest the conceptual connections. At this point perhaps this idea seems academic or overly analytical. But it is truly the case that as you spend some years applying and reflecting on myriad design patterns, you will increasingly come to feel that it's the underlying themes that are important, and the fine details of Adapter or Strategy or whatever will become secondary.



Note that in the *ServicesFactory*, the logic to decide which class to create is resolved by reading in the class name from an external source (for example, via a system property if Java is used) and then dynamically loading the class. This is an example of a partial **data-driven design**. This design achieves Protected Variations with respect to changes in the implementation class of the adapter. Without changing the source code in this factory class, we can create instances of new adapter classes by changing the property value and ensuring that the new class is visible in the Java class path for loading.

## Related Patterns

Factories are often accessed with the Singleton pattern.

## Singleton (GoF)

The *ServicesFactory* raises another new problem in the design: Who creates the factory itself, and how is it accessed?

First, observe that only one instance of the factory is needed within the process. Second, quick reflection suggests that the methods of this factory may need to be called from various places in the code, as different places need access to the adapters for calling on the external services.

Thus, there is a visibility problem: How to get visibility to this single *ServicesFactory* instance?

One solution is pass the *ServicesFactory* instance around as a parameter to wherever a visibility need is discovered for it, or to initialize the objects that need visibility to it, with a permanent reference. This is possible but inconvenient; an alternative is the **Singleton** pattern.

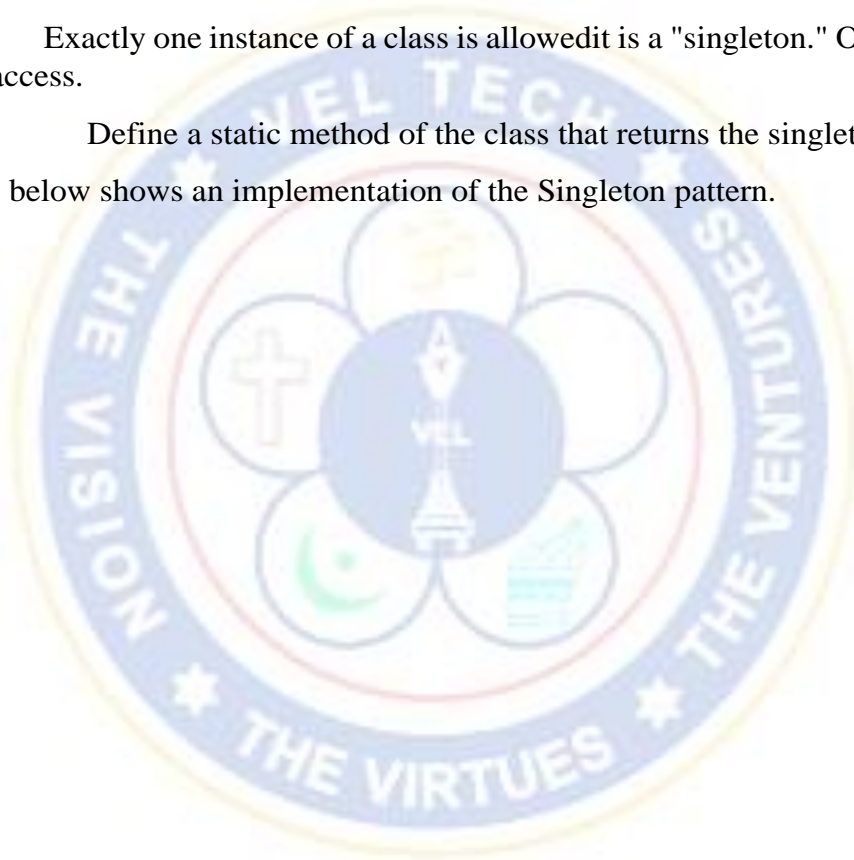
Occasionally, it is desirable to support global visibility or a single access point to a single instance of a class rather than some other form of visibility. This is true for the *ServicesFactory* instance.

Name: **Singleton**

Problem: Exactly one instance of a class is allowed it is a "singleton." Objects need a global and single point of access.

Solution: Define a static method of the class that returns the singleton.  
(advice)

For example, Fig16 below shows an implementation of the Singleton pattern.



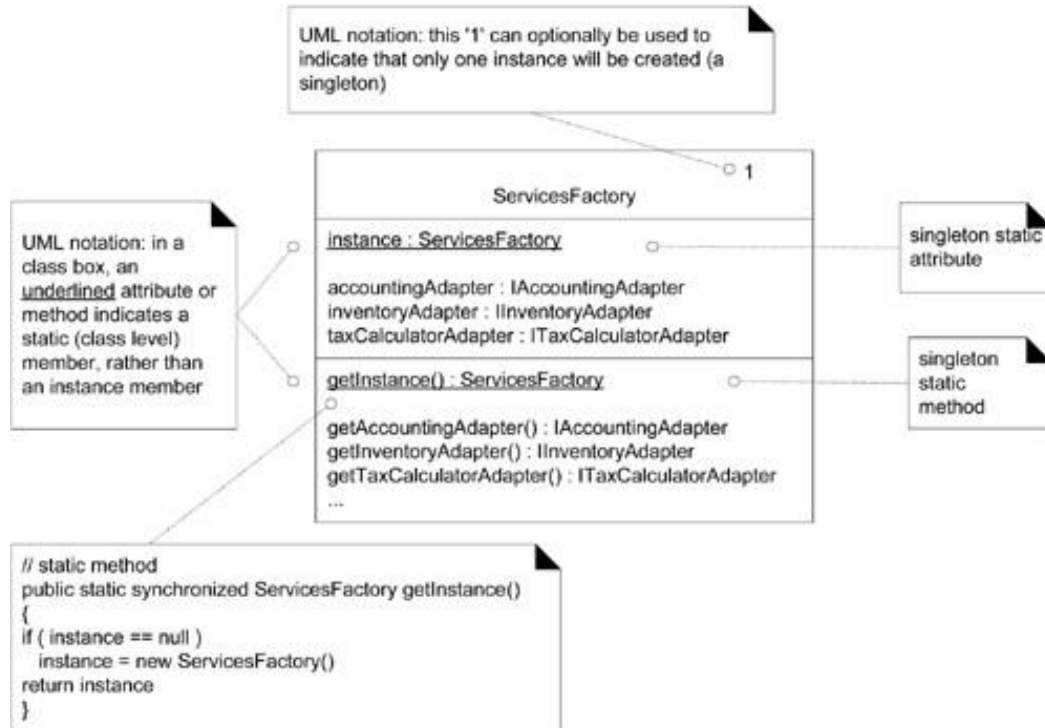


Fig16. The Singleton pattern in the *ServicesFactory* class.

**Applying UML:** Notice how a singleton is illustrated, with a '1' in the top right corner of the name compartment.

Thus, the key idea is that class X defines a static method *getInstance* that itself provides a single instance of X.

With this approach, a developer has global visibility to this single instance, via the static *getInstance* method of the class, as in this example:

```
public class Register
{
public void initialize()
{
... do some work ...
// accessing the singleton Factory via the getInstance call accountingAdapter =
ServicesFactory.getInstance().getAccountingAdapter();
... do some work ...
}
```

**// other methods...**

**} // end of class**

Since visibility to public classes is global in scope (in most languages), at any point in the code, in any method of any class, one can write





## CS6502- OBJECT ORIENTED ANALYSIS AND DESIGN

### UNIT V CODING AND TESTING

Mapping design to code – Testing: Issues in OO Testing – Class Testing – OO Integration Testing – GUI Testing – OO System Testing.

---

#### **1. MAPPING DESIGNS TO CODE**

##### **INTRODUCTION**

##### **Programming and the Development Process**

- The prior design work should not be taken to imply that there is no prototyping or design while programming; modern development tools provide an excellent environment to quickly explore alternate approaches, and some (or even lots) design-while-programming is usually worthwhile.
- However, some developers find that a little forethought with visual modeling before programming is helpful, especially those who are comfortable with visual thinking or diagrammatic languages.
- The creation of code in an object-oriented programming language—such as Java or C#—is not part of OOA/D; it is an end goal. The artifacts created in the UP Design Model provide some of the information
- /D and OO programming—when used with the UP—is that they provide an end-to-end roadmap from requirements through to code. The various artifacts feed into later artifacts in a traceable and useful manner, ultimately culminating in a running application. Necessary to generate the code.

##### ***Creativity and Change During Implementation***

- Some decision-making and creative work was accomplished during design work. It will be seen during the following discussion that the generation of the code— in this example— is a relatively mechanical translation process.
- However, in general, the programming work is not a trivial code generation step— quite the opposite. Realistically, the results generated during design are an incomplete first step; during programming and testing, myriad changes will be made and detailed problems will be uncovered and resolved.
- Done well, the design artifacts will provide a resilient core that scales up with elegance and robustness to meet the new problems encountered during programming. Consequently, expect and plan for change and deviation from the design during programming.

##### ***Code Changes and the Iterative Process***

A strength of an iterative and incremental development process is that the results of a prior iteration can feed into the beginning of the next iteration (see Fig1 below). Thus, subsequent analysis and design results are continually being refined and enhanced from prior implementation work. For example, when the code in iteration N deviates from the design of

iteration N (which it inevitably will), the final design based on the implementation can be input to the analysis and design models of iteration N+1.

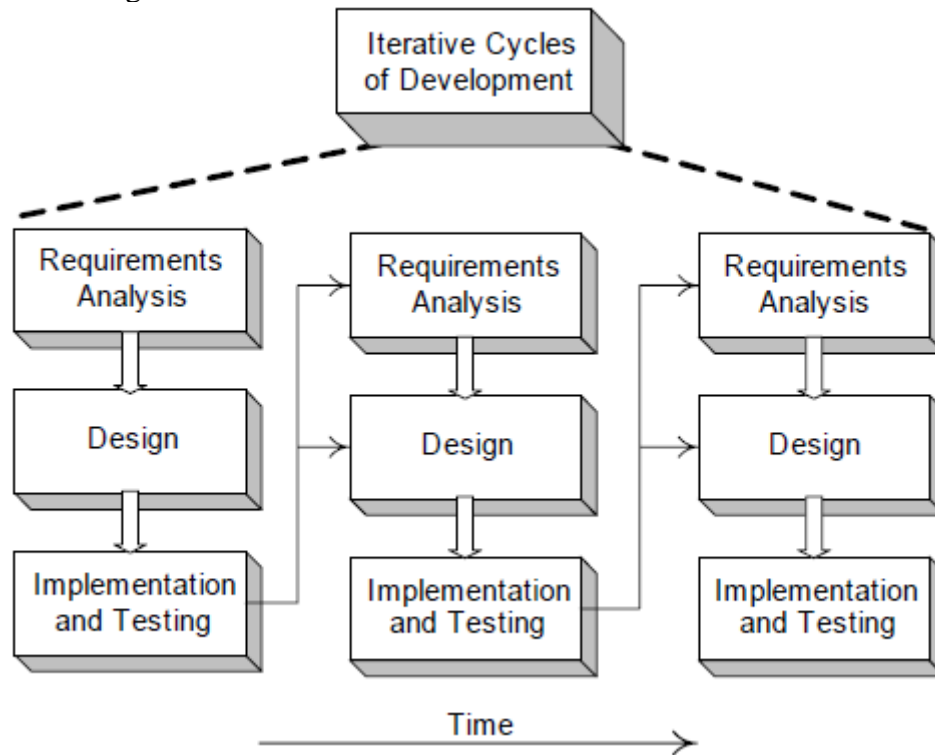


Fig1. Implementation in an iteration influences later design.

An early activity within an iteration is to synchronize the design diagrams; the earlier diagrams of iteration N will not match the final code of iteration N, and they need to be synchronized before being extended with new design results.

### ***Code Changes, CASE Tools, and Reverse-Engineering***

It is desirable for the diagrams generated during design to be semi-automatically updated to reflect changes in the subsequent coding work. Ideally this should be done with a CASE tool that can read source code and automatically generate, for example, package, class, and sequence diagrams.

This is an aspect of **reverse-engineering**—the activity of generating diagrams from source (or sometimes, executable) code.

## **MAPPING DESIGNS TO CODE**

**Explain implementation of models. (Mapping design to code). (May/June 2015, 2016, Nov/Dec 2016, April/May 2017)**

Implementation in an object-oriented programming language requires writing source code for:

- Class and interface definitions
- Method definitions

The following sections discuss their generation in Java (as a typical case).

### **Creating Class Definitions from DCDs**

At the very least, DCDs depict the class or interface name, superclasses, method signatures, and simple attributes of a class. This is sufficient to create a basic class definition in an object-oriented programming language.

### ***Defining a Class with Methods and Simple Attributes***

From the DCD, a mapping to the basic attribute definitions (simple Java instance fields) and method signatures for the Java definition of *SalesLineItem* is straightforward, as shown in Fig2 below.

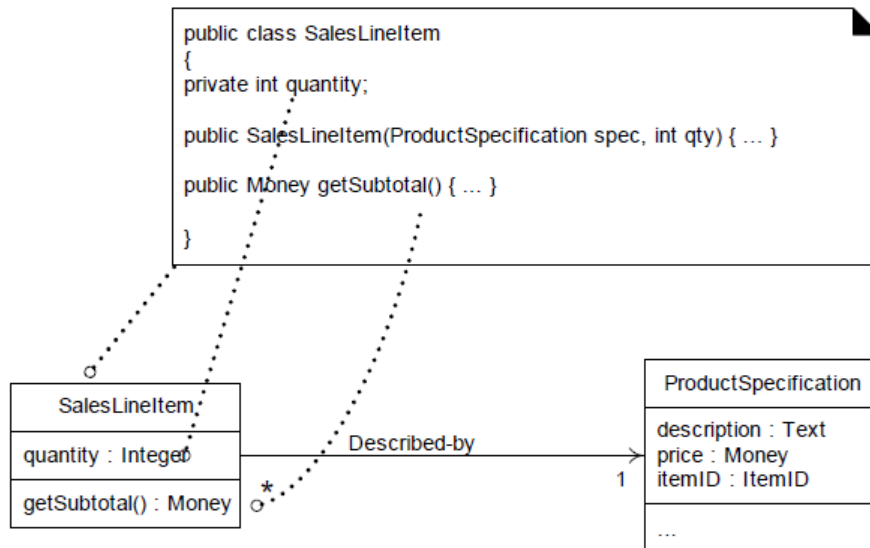


Fig2. SalesLineItem in Java.

Note the addition in the source code of the Java constructor *SalesLineItem(...)*. It is derived from the *create(spec, qty)* message sent to a *SalesLineItem* in the *enterItem* interaction diagram. This indicates, in Java, that a constructor supporting these parameters is required.

The *create* method is often excluded from the class diagram because of its commonality and multiple interpretations, depending on the target language.

### ***Adding Reference Attributes***

- A **reference attribute** is an attribute that refers to another complex object, not to a primitive type such as a String, Number, and so on.
- The reference attributes of a class are suggested by the associations and navigability in a class diagram.
- For example, a *SalesLineItem* has an association to a *ProductSpecification*, with navigability to it. It is common to interpret this as a reference attribute in class *SalesLineItem* that refers to a *ProductSpecification* instance (see Fig3 below).
- In Java, this means that an instance field referring to a *ProductSpecification* instance is suggested.

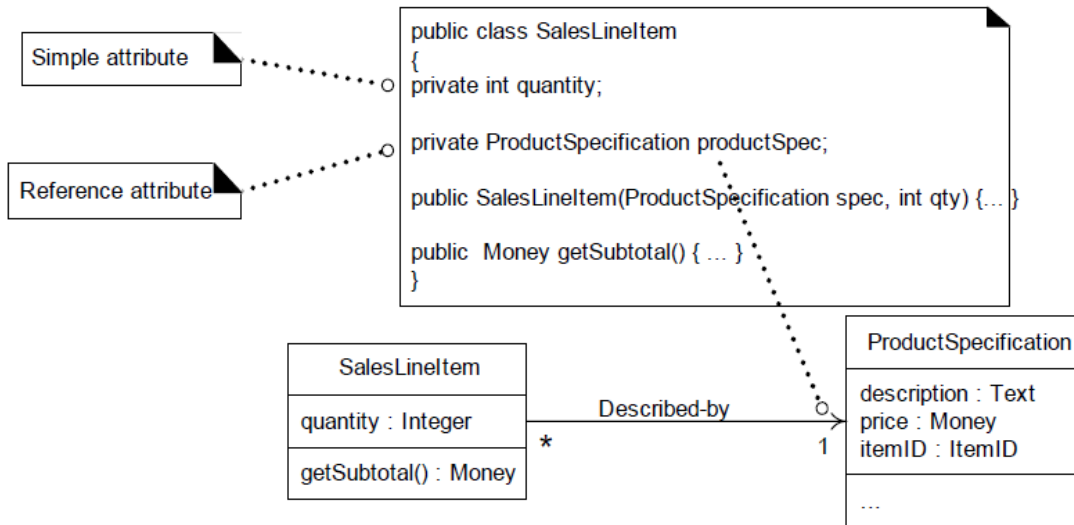


Fig3. Adding reference attributes.

Note that reference attributes of a class are often implied, rather than explicit, in a DCD.

For example, although we have added an instance field to the Java definition of *SalesLineItem* to point to a *ProductSpecification*, it is not explicitly declared as an attribute in the attribute section of the class box. There is a *suggested* attribute visibility—indicated by the association and navigability—which is explicitly defined as an attribute during the code generation phase.

### Reference Attributes and Role Names

The next iteration will explore the concept of role names in static structure diagrams.

Each end of an association is called a role. Briefly, a **role name** is a name that identifies the role and often provides some semantic context as to the nature of the role.

If a role name is present in a class diagram, use it as the basis for the name of the reference attribute during code generation, as shown in Fig4 below.

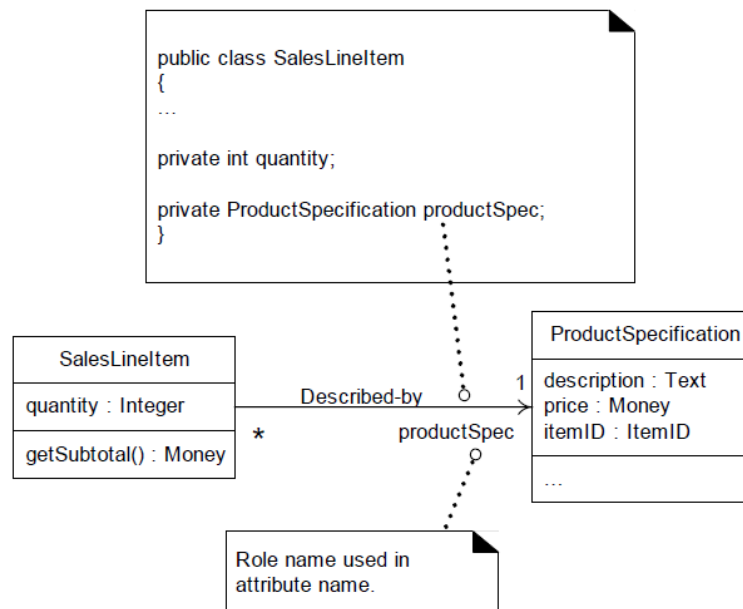


Fig4. Role names may be used to generate instance variable names.

**Mapping Attributes**

The *Sale* class illustrates that in some cases one must consider the mapping of attributes from the design to the code in different languages. Figure below illustrates the problem and its resolution.

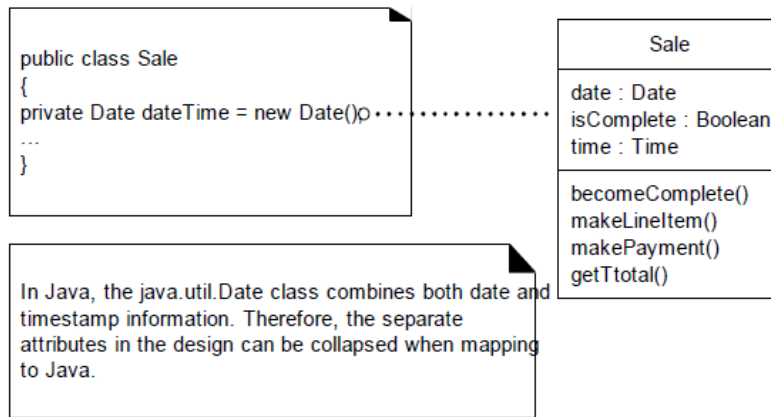


Fig5. Mapping date and time to Java.

**Creating Methods from Interaction Diagrams**

An interaction diagram shows the messages that are sent in response to a method invocation. The sequence of these messages translates to a series of statements in the method definition. The *enterItem* interaction diagram in Figure below illustrates the Java definition of the *enterItem* method.

In this example, the *Register* class will be used. A Java definition is shown in Fig6 below.

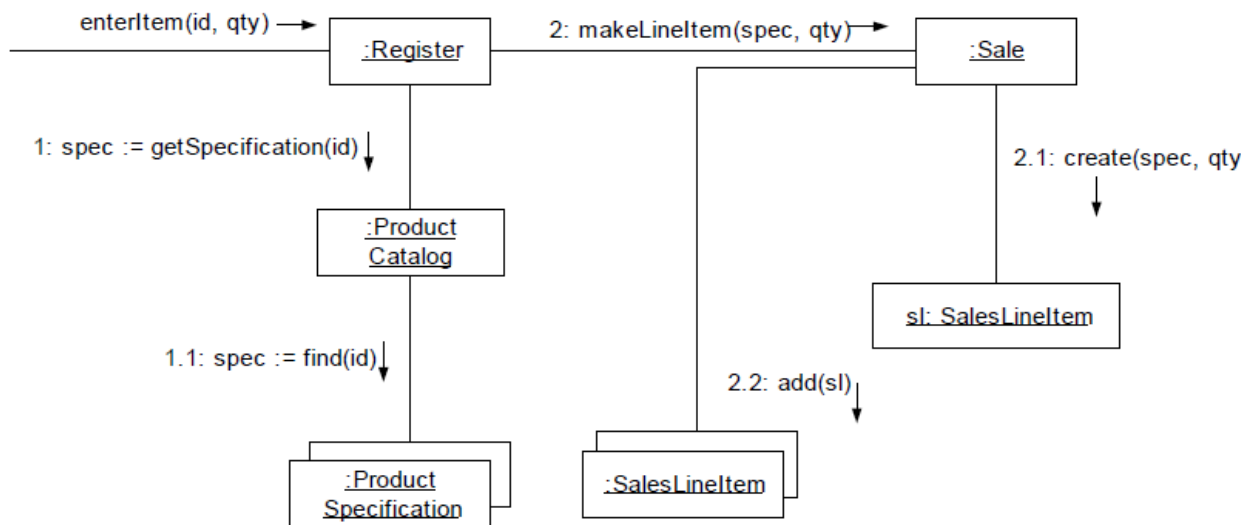


Fig6.The enterItem interaction diagram.

**The Register-enterItem Method**



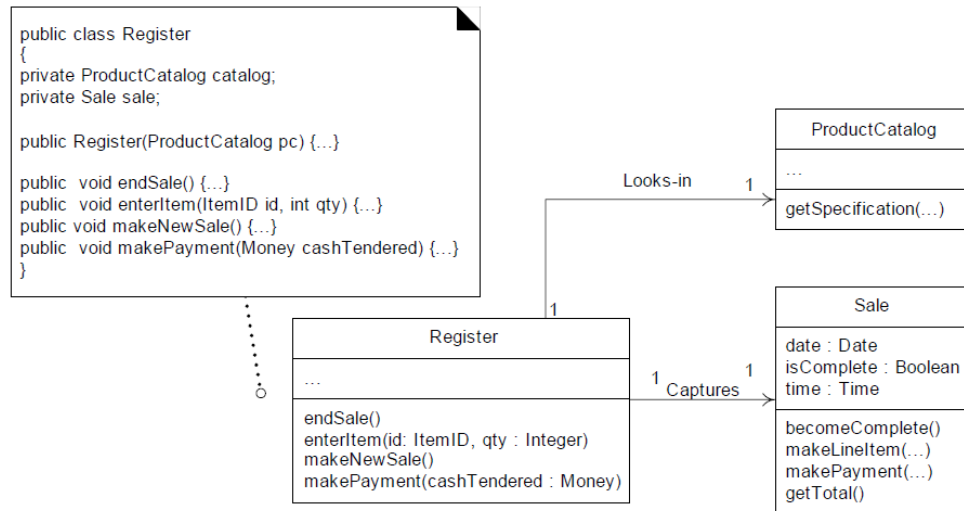


Fig7. The Register class.

The *enterItem* message is sent to a *Register* instance; therefore, the *enterItem* method is defined in class *Register*.

```
public void enterItem ( ItemID itemID, int qty)
```

**Message 1:** A *getSpecification* message is sent to the *ProductCatalog* to retrieve a *ProductSpecification*.

```
Product Specification spec = catalog.getSpecification( itemID );
```

**Message 2:** The *makeLineItem* message is sent to the *Sale*.

```
sale .makeLineItem( spec, qty);
```

In summary, each sequenced message within a method, as shown on the interaction diagram, is mapped to a statement in the Java method. The complete *enterItem* method and its relationship to the interaction diagram is shown in Fig8 below.

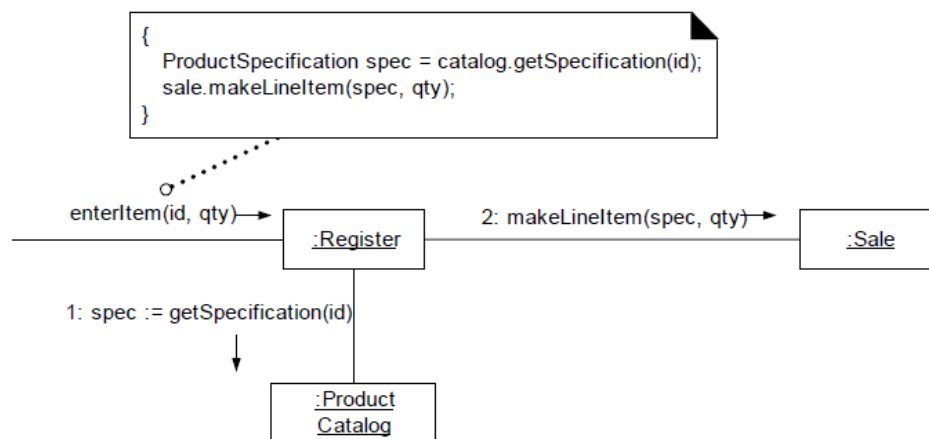


Fig8. The enterItem method.

### Container/Collection Classes in Code

It is often necessary for an object to maintain visibility to a group of other objects; the need for this is usually evident from the multiplicity value in a class diagram—it may be greater than one. For example, a *Sale* must maintain visibility to a group of *SalesLineItem* instances, as shown in Fig9 below.

In OO programming languages, these relationships are often implemented with the introduction of an intermediate container or collection. The one-side class defines a reference attribute pointing to a container/collection instance, which contains instances of the many-side class.

### Exceptions and Error Handling

Exception handling has been ignored so far in the development of a solution.

This was intentional to focus on the basic questions of responsibility assignment and object design. However, in application development, it is wise to consider exception handling during design work, and certainly during implementation. Briefly, in the UML, exceptions are illustrated as asynchronous messages in interaction diagrams.

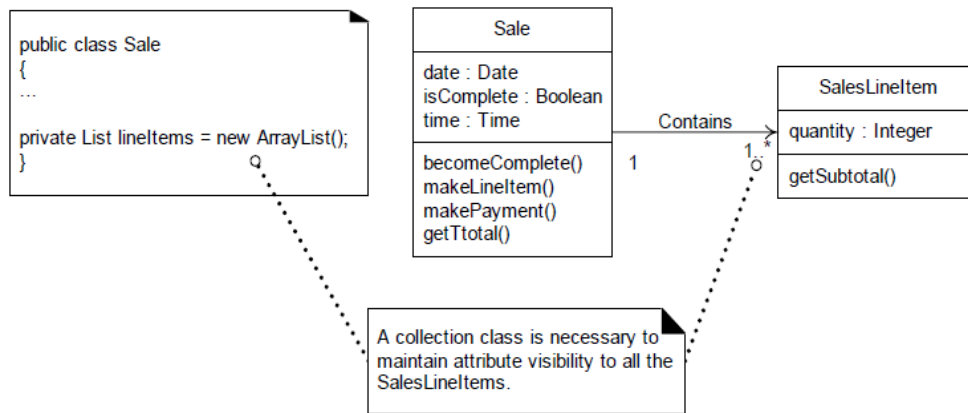


Fig9. Adding a collection.

### Defining the Sale--makeLineItem Method

As a final example, the *makeLineItem* method of class *Sale* can also be written by inspecting the *enterItem* collaboration diagram. An abridged version of the interaction diagram, with the accompanying Java method, is shown in Fig10 below.

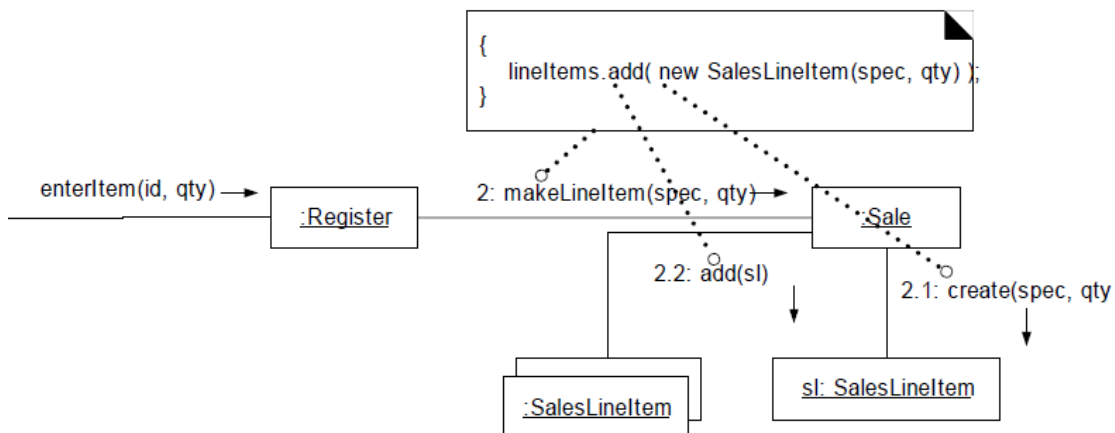


Fig10. Sale-makeLineItem method.

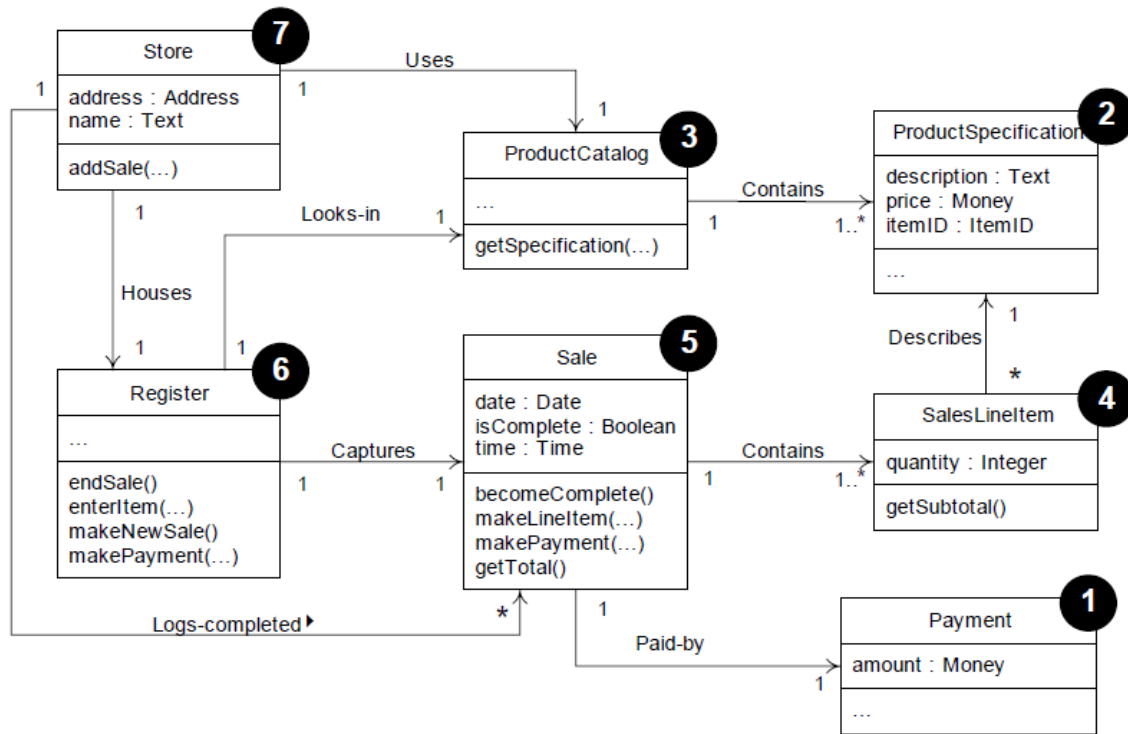


Fig11. Possible order of class implementation and testing.

### Order of Implementation

Classes need to be implemented (and ideally, fully unit tested) from least-coupled to most-coupled (see Figure above). For example, possible first classes to implement are either *Payment* or *ProductSpecification*; next are classes only dependent on the prior implementations—*ProductCatalog* or *SalesLineItem*.

### Test-First Programming

An excellent practice promoted by the Extreme Programming (XP) method and applicable to the UP (as most XP practices are), is **test-first programming**.

In this practice, unit testing code is written *before* the code to be tested, and the developer writes unit testing code for *all* production code. The basic rhythm is to write a little test code, then write a little production code, make it pass the test, then write some more test code, and so forth.

## ORDER OF IMPLEMENTATION

### IMPLEMENTATION MODEL: MAPPING DESIGNS TO CODE

#### Advantages include:

- **The unit tests actually get written**—Human (or at least programmer) nature is such that avoidance of writing unit tests is very common, if left as an afterthought.

- **Programmer satisfaction**—If a developer writes the production code, informally debugs it, and then as an afterthought adds unit tests, it does not feel very satisfying. However, if the tests are written first, and then production code is created and refined to pass the tests, there is some feeling of accomplishment—of passing a test. The psychological aspects of development can't be ignored—programming is a human endeavor.

- **Clarification of interface and behavior**—Often, the exact public interface and behavior of a class is not perfectly clear until programming it. By writing the unit test for it first, one clarifies the design of the class.

- **Provable verification**—Obviously, having hundreds or thousands of unit tests provides some meaningful verification of correctness.

- **The confidence to change things**—In test-first programming, there are hundreds or thousands of unit tests, and a unit test class for each production class. When a developer needs to change existing code—written by themselves or others—there is a unit test suite that can be run, providing immediate feedback if the change caused an error.

As an example, a popular, simple and free unit testing framework is JUnit for Java. Suppose we are using JUnit and test-first programming to create the *Sale* class. *Before* programming the *Sale* class, we write a unit testing method in a *SaleTest* class that does the following:

1. Set up a new sale.
2. Add some line items to it.
3. Ask for the total, and verify it is the expected value.

For example:

```
public class SaleTest extends TestCase {
// ...
public void testTotal() {
// set up the test
Money total = new Money (7 . 5 );
Money price = new Money (2 . 5 );
ItemID id = new ItemID (1 );
ProductSpecification spec;
spec = new ProductSpecification( id, price, "product 1" );
Sale sale = new SaleO;
// add the items
sale.makeLineltern( spec, 1 );
sale.makeLineltern(spec, 2 );
// verify the total is 7 . 5
assertEquals( sale.getTotal(), total); } }
```

Only after this *SaleTest* class is created do we then write the *Sale* class to pass this test. However, not all unit testing methods need to be written beforehand. A developer writes one testing method, then the production code to satisfy it, then another testing method, and so on.

## **2. TESTING: ISSUES IN OO TESTING**

### **Explain the issues involved in OO testing.(April/May 2017)**

#### **What is a unit in an object orientated system?**

- Traditional systems define a unit as the smallest component that can be compiled and executed.
- Units are normally a component which in theory is only ever assigned to one programmer.
- Two options for selecting units in object orientated systems:

- Treat each class as a unit
- Treat each method within a class as a unit.

### **Advantages of OO Unit Testing**

- Once a class is testing thoroughly it can be reused without being unit tested again.
- UML class state charts can help with selection of test cases for classes.
- Classes easily mirror units in traditional software testing

### **Disadvantages of OO Unit Testing**

- Classes obvious unit choice, but they can be large in some applications
- Problems dealing with polymorphism and inheritance.

### **Composition Issues**

- Objective of OO is to facilitate easy code reuse in the form of classes.
- To allow this each class has to be rigorously unit tested.
- Due to classes potentially used in unforeseeable ways when composed in new systems I

**Example:** A XML parser for a web browser.

- Classes must be created in a way promoting loose coupling and strong cohesion
- Encapsulation Issues
- Encapsulation requires that classes are only aware of their own properties, and are able to operate independently.
- If unit testing is performed well, the integration testing becomes more important.
- If you do not have access to source code then structural testing can be impossible.
- If you violate encapsulation for testing purposes, then the validity of test could be questionable

### **Implication of Inheritance and Polymorphism**

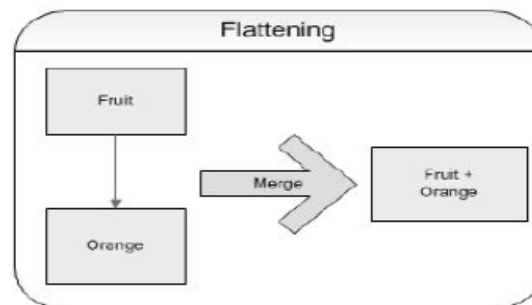
- Inheritance is an important part of the object oriented paradigm.
- Unit testing a class with a super class can be impossible to do without the super classes methods/variables

### **One solution is Flattening**

- Merge the super class, and the class under test so all methods/variables are available.
- Solves initial unit test problems.

#### **Problems:**

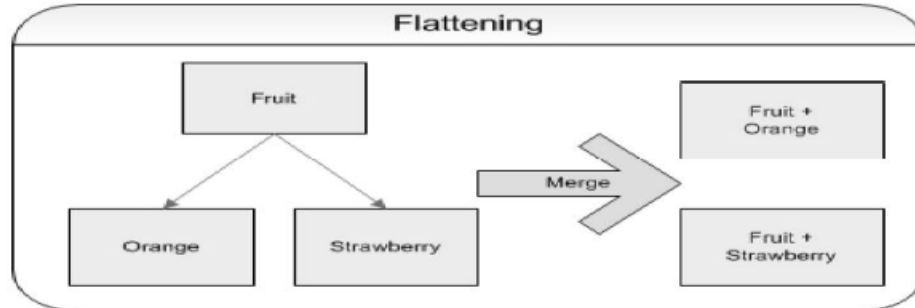
- The class won't be flattened in the final product so potential issues may still arise.
- Complicated when dealing with multiple inheritance





### Polymorphism Issues

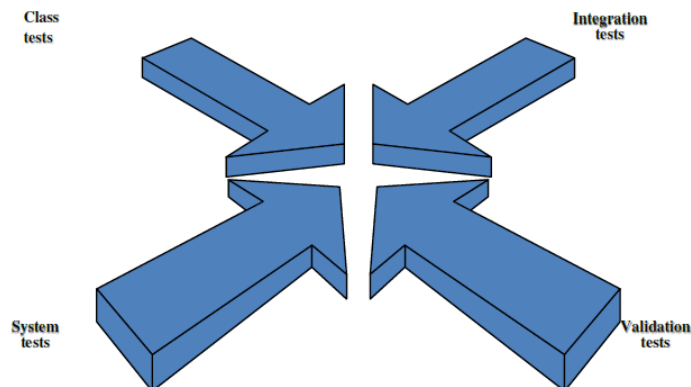
- Repeatedly testing same methods
- Time can then be wasted if not addressed
- Potentially can be avoided and actually save time



### 3. CLASS (UNIT) TESTING

#### Discuss on class Testing with its impacts.

#### Testing OO Code



- Smallest testable unit is the encapsulated class
- Test each operation as part of a class hierarchy because its class hierarchy defines its context of use

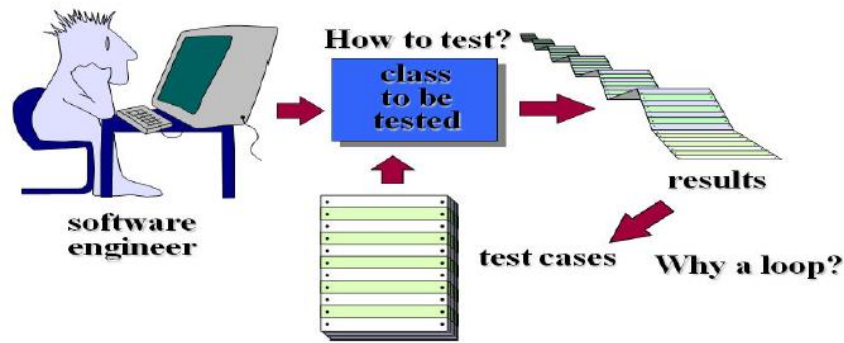
#### **Approach:**

- Test each method (and constructor) within a class
- Test the state behavior (attributes) of the class between methods

#### *How is class testing different from conventional testing?*

- Conventional testing focuses on input-process-output, whereas class testing focuses on each method, then designing sequences of methods to exercise states of a class
- But white-box testing can still be applied

#### **Class Testing Process**



### Class Test Case Design

#### 1. Identify each test case uniquely

- Associate test case explicitly with the class and/or method to be tested

#### 2. State the purpose of the test

#### 3. Each test case should contain:

- A list of messages and operations that will be exercised as a consequence of the test
- A list of exceptions that may occur as the object is tested
- A list of external conditions for setup (i.e., changes in the environment external to the software that must exist in order to properly conduct the test)
- Supplementary information that will aid in understanding or implementing the test

- Automated unit testing tools facilitate these requirements

### Challenges of Class Testing

#### • Encapsulation:

Difficult to obtain a snapshot of a class without building extra methods which display the classes' state

#### • Inheritance and polymorphism:

- o Each new context of use (subclass) requires re-testing because a method may be implemented differently (polymorphism).
- o Other unaltered methods within the subclass may use the redefined method and need to be tested

#### • White box tests:

o Basis path, condition, data flow and loop tests can all apply to individual methods, but don't test interactions between methods

### Testing Methods Applicable at the Class Level

**1. Random testing** - requires large numbers data permutations and combinations, and can be inefficient

- Identify operations applicable to a class
- Define constraints on their use
- Identify a minimum test sequence
- Generate a variety of random test sequences.

### Example:

- An account class in a banking application has open, setup, deposit, withdraw, balance, summarize and close methods
- The account must be opened first and closed on completion

- Open – setup – deposit – withdraw – close
  - Open – setup – deposit –\* [deposit | withdraw | balance | summarize] – withdraw – close. Generate random test sequences using this template
- 2. Partition testing** - reduces the number of test cases required to test a class
- **state-based partitioning** - tests designed in way so that operations that cause state changes are tested separately from those that do not.
  - **attribute-based partitioning** - for each class attribute, operations are classified according to those that use the attribute, those that modify the attribute, and those that do not use or modify the attribute
  - **category-based partitioning** - operations are categorized according to the function they perform: initialization, computation, query, termination
- 3. Fault-based testing**
- best reserved for operations and the class level
  - uses the inheritance structure
  - tester examines the OOA model and hypothesizes a set of plausible defects that may be encountered in operation calls and message connections and builds appropriate test cases
  - misses incorrect specification and errors in subsystem interactions

#### **4. INTEGRATION TESTING**

##### **Discuss on OO integration testing with its impacts. (Nov/Dec 2015, May/June 2016)**

OO does not have a hierarchical control structure so conventional top-down and bottom up integration tests have little meaning.

##### **Kinds of integration testing:**

- **big bang testing (5 facts)** - An inappropriate approach to integration testing in which you take the entire integrated system and test it as a unit
- **incremental testing (2 kinds, 3 facts)** - A integration testing strategy in which you test subsystems in isolation, and then continue testing as you integrate more and more subsystems.

##### **Integration applied three different incremental strategies:**

- Thread-based testing: integrates classes required to respond to one input or event
- Use-based testing: integrates classes required by one use case
- Cluster testing: integrates classes required to demonstrate one collaboration

##### **Inter-Class Test Case Design**

- Test case design becomes more complicated as integration of the OO system begins – testing of collaboration between classes
- **Multiple class testing**
  - for each client class use the list of class operators to generate random test sequences that send messages to other server classes
  - for each message generated determine the collaborator class and the corresponding server object operator
  - for each server class operator (invoked by a client object message) determine the message it transmits

- for each message, determine the next level of operators that are invoked and incorporate them into the test sequence
- **Tests derived from behavior models**
  - Use the state transition diagram (STD) as a model that represent the dynamic behavior of a class.
  - test cases must cover all states in the STD
  - breadth first traversal of the state model can be used (test one transition at a time and only make use of previously tested transitions when testing a new transition)
  - test cases can also be derived to ensure that all behaviors for the class have been adequately exercised

### Testing Methods Applicable at Inter-Class Level

- **Cluster Testing**
  - Is concerned with integrating and testing clusters of cooperating objects
  - Identify clusters using knowledge of the operation of objects and the system features that are implemented by these clusters
  - **Approaches to Cluster Testing**
    - Use-case or scenario testing
      - Testing is based on a user interactions with the system
      - Has the advantage that it tests system features as experienced by users
    - Thread testing – tests the systems response to events as processing threads through the system
    - Object interaction testing – tests sequences of object interactions that stop when an object operation does not call on services from another object
- **Use Case/Scenario-based Testing**
  - Based on
    - use cases
    - corresponding sequence diagrams
  - Identify scenarios from use-cases and supplement these with interaction diagrams that show the objects involved in the scenario
  - Concentrates on (functional) requirements
    - Every use case
    - Every fully expanded extension (<<extend>>) combination
    - Every fully expanded uses (<<uses>>) combination
    - Tests normal as well as exceptional behavior
  - A scenario is a path through sequence diagram
  - Many different scenarios may be associated with a sequence diagram
  - using the user tasks described in the use-cases and building the test cases from the tasks and their variants
  - uncovers errors that occur when any actor interacts with the OO software
  - concentrates on what the use does, not what the product does
  - you can get a higher return on your effort by spending more time on reviewing the use-cases as they are created, than spending more time on use-case testing



## 5. GUI TESTING

### Discuss on with GUI Testing its impacts. (April/May 2017)

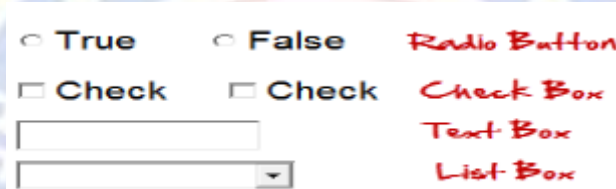
GUI testing is the process of ensuring proper functionality of the graphical user interface (GUI) for a given application and making sure it conforms to its written specifications. In addition to functionality, GUI testing evaluates design elements such as layout, colors, fonts, font sizes, labels, text boxes, text formatting, captions, buttons, lists, icons, links and content.

GUI testing processes can be either manual or automatic, and are often performed by third -party companies, rather than developers or end users. GUI testing can require a lot of programming and is time consuming whether manual or automatic.

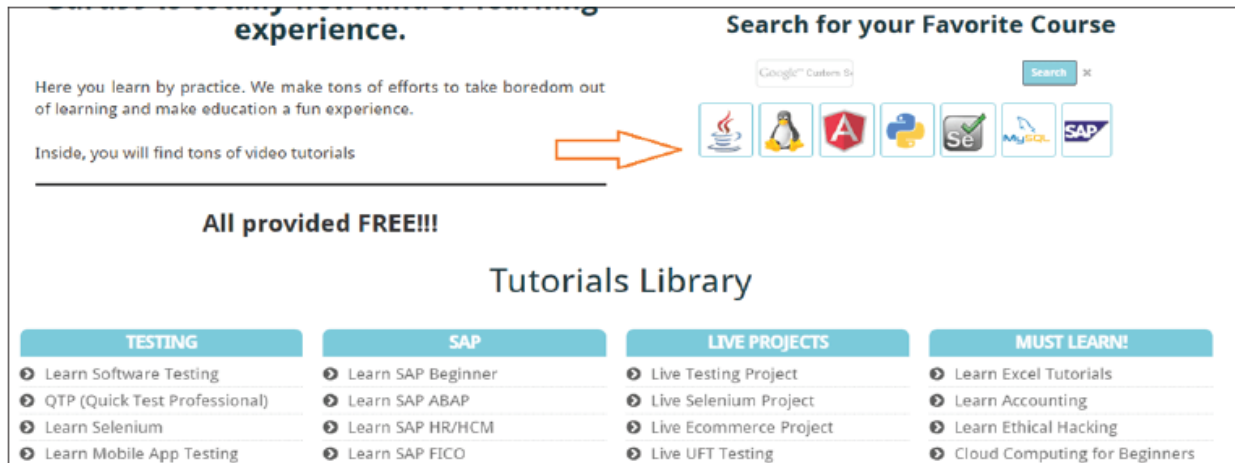
**There are two types of interfaces in a computer application.**

- Command Line Interface is where you type text and computer responds to that command.
- GUI stands for Graphical User Interface where you interact with the computer using images rather than text.

Following are the GUI elements which can be used for interaction between the user and application:



In the below example, if we have to do GUI testing we first check that the images should be completely visible in different browsers. Also, the links are available, and the button should work when clicked. Also, if the user resizes the screen, neither images nor content should shrink or crop or overlap.



experience. Search for your Favorite Course

Here you learn by practice. We make tons of efforts to take boredom out of learning and make education a fun experience.

Inside, you will find tons of video tutorials

All provided FREE!!!

### Tutorials Library

| TESTING                         | SAP                  | LIVE PROJECTS            | MUST LEARN!                     |
|---------------------------------|----------------------|--------------------------|---------------------------------|
| • Learn Software Testing        | • Learn SAP Beginner | • Live Testing Project   | • Learn Excel Tutorials         |
| • QTP (Quick Test Professional) | • Learn SAP ABAP     | • Live Selenium Project  | • Learn Accounting              |
| • Learn Selenium                | • Learn SAP HR/HCM   | • Live Ecommerce Project | • Learn Ethical Hacking         |
| • Learn Mobile App Testing      | • Learn SAP FICO     | • Live UFT Testing       | • Cloud Computing for Beginners |

### Need for GUI Testing

- A user doesn't have any knowledge about XYZ software/Application. It is the UI of the Application which decides that a user is going to use the Application further or not.
- A normal User first observes the design and looks of the Application/Software and how easy it is for him to understand the UI. If a user is not comfortable with the Interface or find Application complex to understand he would never going to use that Application Again.



- That's why, GUI is a matter for concern, and proper testing should be carried out in order to make sure that GUI is free of Bugs.

### What do you check in GUI Testing?

The following checklist will ensure detailed GUI Testing.

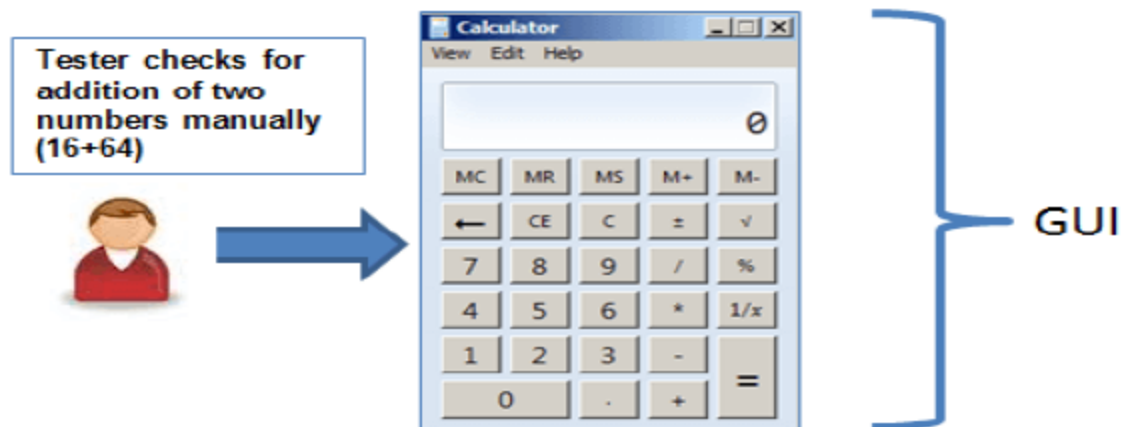
- Check all the GUI elements for size, position, width, length and acceptance of characters or numbers. For instance, you must be able to provide inputs to the input fields.
- Check you can execute the intended functionality of the application using the GUI
- Check Error Messages are displayed correctly
- Check for Clear demarcation of different sections on screen
- Check Font used in application is readable
- Check the alignment of the text is proper
- Check the Color of the font and warning messages is aesthetically pleasing
- Check that the images have good clarity
- Check that the images are properly aligned
- Check the positioning of GUI elements for different screen resolution.

### Approach of GUI Testing

GUI testing can be done through three ways:

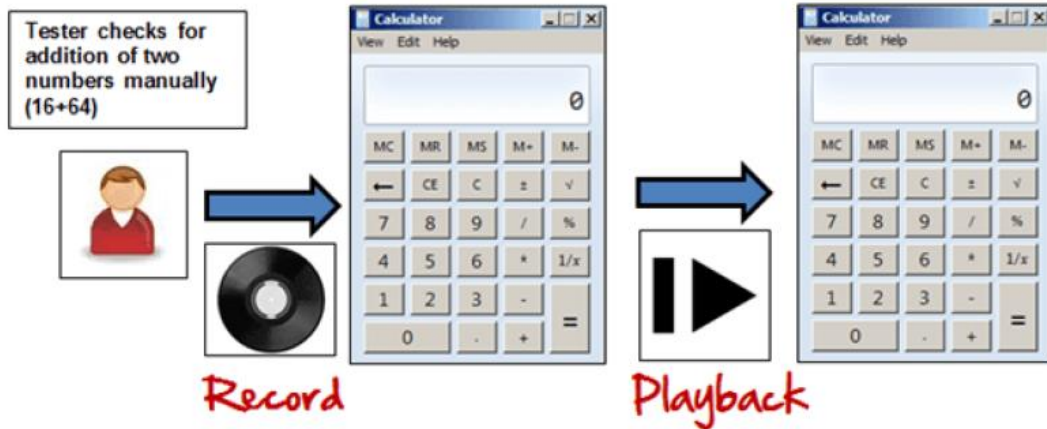
#### Manual Based Testing

Under this approach, graphical screens are checked manually by testers in conformance with the requirements stated in the business requirements document.

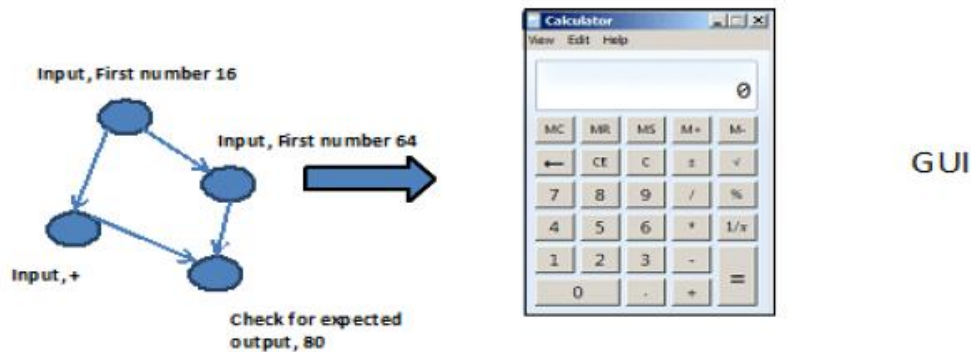


#### Record and Replay

GUI testing can be done using automation tools. This is done in 2 parts. During Record , test steps are captured by the automation tool. During playback, the recorded test steps are executed on the Application Under Test. Example of such tools - QTP .



### Model Based Testing



### Model Based Testing

### GUI Testing Test Cases

#### GUI Testing basically involves

1. Testing the size, position, width, height of the elements.
2. Testing of the error messages that are getting displayed.
3. Testing the different sections of the screen.
4. Testing of the font whether it is readable or not.
5. Testing of the screen in different resolutions with the help of zooming in and zooming out like 640 x 480, 600x800, etc.
6. Testing the alignment of the texts and other elements like icons, buttons, etc. are in proper place or not.
7. Testing the colors of the fonts.
8. Testing the colors of the error messages, warning messages.
9. Testing whether the image has good clarity or not.
10. Testing the alignment of the images.
11. Testing of the spelling.
12. The user must not get frustrated while using the system interface.
13. Testing whether the interface is attractive or not.
14. Testing of the scrollbars according to the size of the page if any.
15. Testing of the disabled fields if any.
16. Testing of the size of the images.

17. Testing of the headings whether it is properly aligned or not.
18. Testing of the color of the hyperlink.

### Challenges in GUI Testing

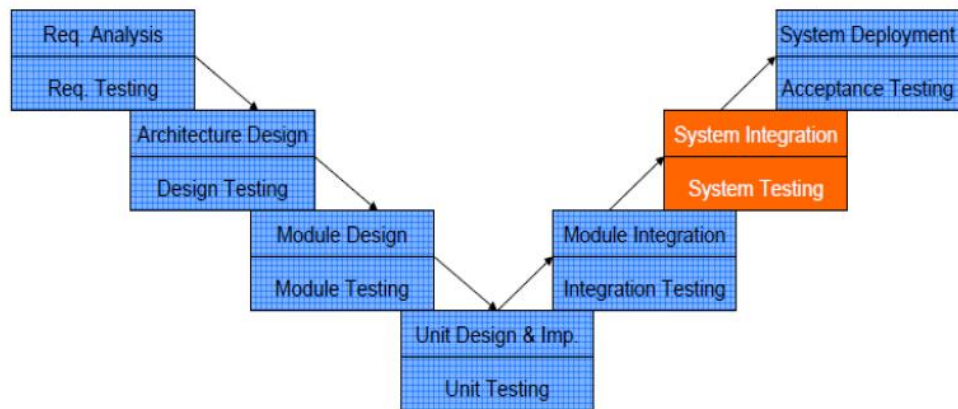
The most common problem comes while doing regression testing is that the application GUI changes frequently. It is very difficult to test and identify whether it is an issue or enhancement. The problem manifests when you don't have any documents regarding GUI changes.

## 6. SYSTEM TESTING

Discuss on with OO system testing its impacts (Nov/Dec 2014, May/june 2016, April/May 2017)

### System Testing

- Tests the system as a whole
- Concerned with what happens
- Not how it happens
- Black box



### Types of System Testing:

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

### Functional 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 new test cases have to be developed as well.

### Structure 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.

### Performance Testing

Goal: Try to break the subsystems

- Test how the system behaves when overloaded.
- Can bottlenecks be identified? (First candidates for redesign in the next iteration)
- Try unusual orders of execution
- Call a receive() before send()
- Check the system's response to large volumes of data
- If the system is supposed to handle 1000 items, try it with 1001 items.
- What is the amount of time spent in different use cases?
- Are typical cases executed in a timely fashion?

### Types of Performance Testing

- **Recovery testing:** how well and quickly does the system recover from faults
- **Security testing:** verify that protection mechanisms built into the system will protect from unauthorized access (hackers, disgruntled employees, fraudsters)
- **Stress testing:** place abnormal load on the system
- **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
- **Timing testing:** Evaluate response times and time to perform a function
- **Environmental test** - Test tolerances for heat, humidity, motion
- **Quality testing:** - Test reliability, maintainability & availability
- **Human factors testing:** Test with end users

### Acceptance Testing

Goal: Demonstrate system is ready for operational use

- Choice of tests is made by client
- Many tests can be taken from integration testing
- Acceptance test is performed by the client, not by the developer.

#### 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 environment.