

UNIT 1 UML DIAGRAMS

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

INTRODUCTION TO OOAD

ANALYSIS

Analysis is a creative activity or an investigation of the problem and requirements.

Eg. To develop a Banking system

Analysis: How the system will be used?

Who are the users?

What are its functionalities?

DESIGN

Design is to provide a conceptual solution that satisfies the requirements of a given problem.

Eg. For a Book Bank System

Design: Bank(Bank name, No of Members, Address)

Student(Membership No,Name,Book Name, Amount Paid)

OBJECT ORIENTED ANALYSIS (OOA)

Object Oriented Analysis is a process of identifying classes that plays an important role in achieving system goals and requirements.

Eg. For a Book Bank System, Classes or Objects identified are Book-details,

Student-details, Membership-Details.

OBJECT ORIENTED DESIGN (OOD)

Object Oriented Design is to design the classes identified during analysis phase and to provide the relationship that exists between them that satisfies the requirements.

Eg. Book Bank System

Class name Book-Bank (Book-Name, No-of-Members, Address)

Student (Name, Membership No, Amount-Paid)

OBJECT ORIENTED ANALYSIS AND DESIGN (OOAD)

- OOAD is a Software Engineering approach that models an application by a set of Software Development Activities.

- OOAD emphasis on identifying, describing and defining the software objects and shows how they collaborate with one another to fulfill the requirements by applying the object oriented paradigm and visual modeling throughout the development life cycles.

UNIFIED PROCESS (UP)

The Unified Process has emerged as a popular iterative software development process for building object oriented systems. The Unified Process (UP) combines commonly accepted best practices, such as an iterative lifecycle and risk-driven development, into a cohesive and well-documented description. The best-known and extensively documented refinement of the Unified Process is the Rational Unified Process (RUP).

Reasons to use UP

- UP is an iterative process
- UP practices provide an example structure to talk about how to do, and how to learn OOA/D.

Best Practices and Key Concepts in UP

- Tackle high-risk and high-value issues in early iterations
- Engage users continuously for evaluation, feedback, and requirements
- Build a cohesive, core architecture in early iterations
- Apply use cases
- Provides visual modeling using UML
- Practice change request and configuration management.

UP PHASES

There are 4 phases in Unified Process,

1. Inception
2. Elaboration
3. Construction
4. Transition

INCEPTION

Inception is the initial stage of the project. Inception is not a requirements phase but it is a feasibility phase where complete investigation takes place to support a decision to continue or stop. It deals with

- Approximate vision
- Business case
- Scope
- Vague estimates

ELABORATION

In Elaboration phase the project team is expected to capture a healthy majority of the system requirements. It deals with

- Refined vision,
- Iterative implementation of the core architecture,
- Resolution of high risks,
- Identification of most requirements and scope,
- Realistic estimates.

CONSTRUCTION

Construction phase encompasses on iterative implementation of the remaining lower risk and easier elements, and preparation for deployment.

TRANSITION

Transition phase focus on releasing the final product to the customers for usability.

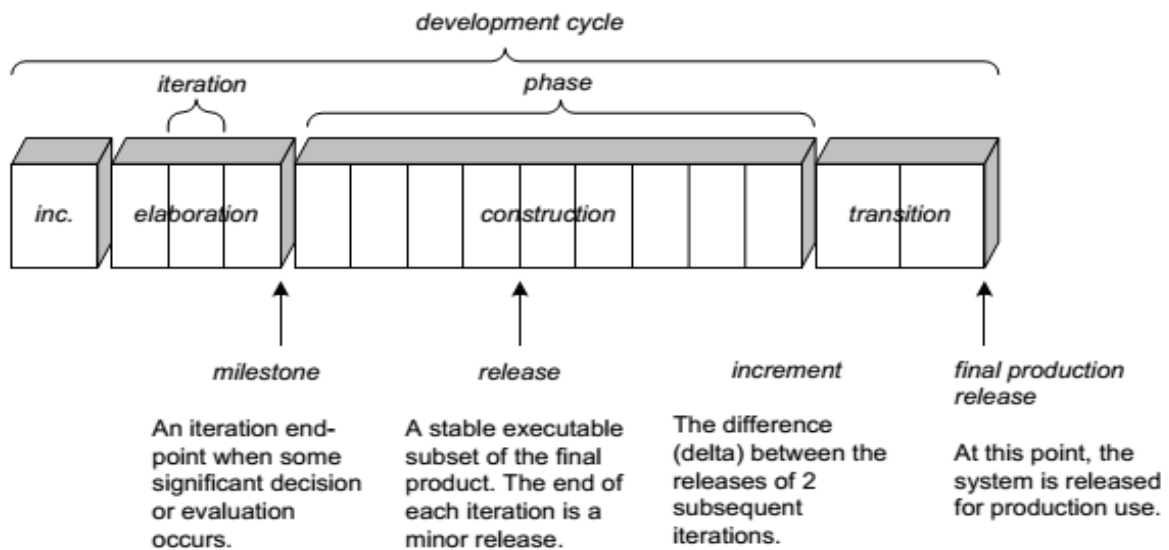


Fig: Phases of UP

UP DISCIPLINES

- UP describes work activities such as writing a use case within disciplines a set of activities and related artifacts in one subject area within requirement analysis.
- Artifact-any work such as code, web graphics, database schema, text documents, diagrams, models etc.

Several UP Disciplines

1. **Business Modeling-** Domain Model artifact to visualize concepts in the application domain.
2. **Requirements-** use case model and specification artifacts to capture functional and non-functional requirements.
3. **Design-** All aspects of design, including overall architecture, objects, databases, networking.

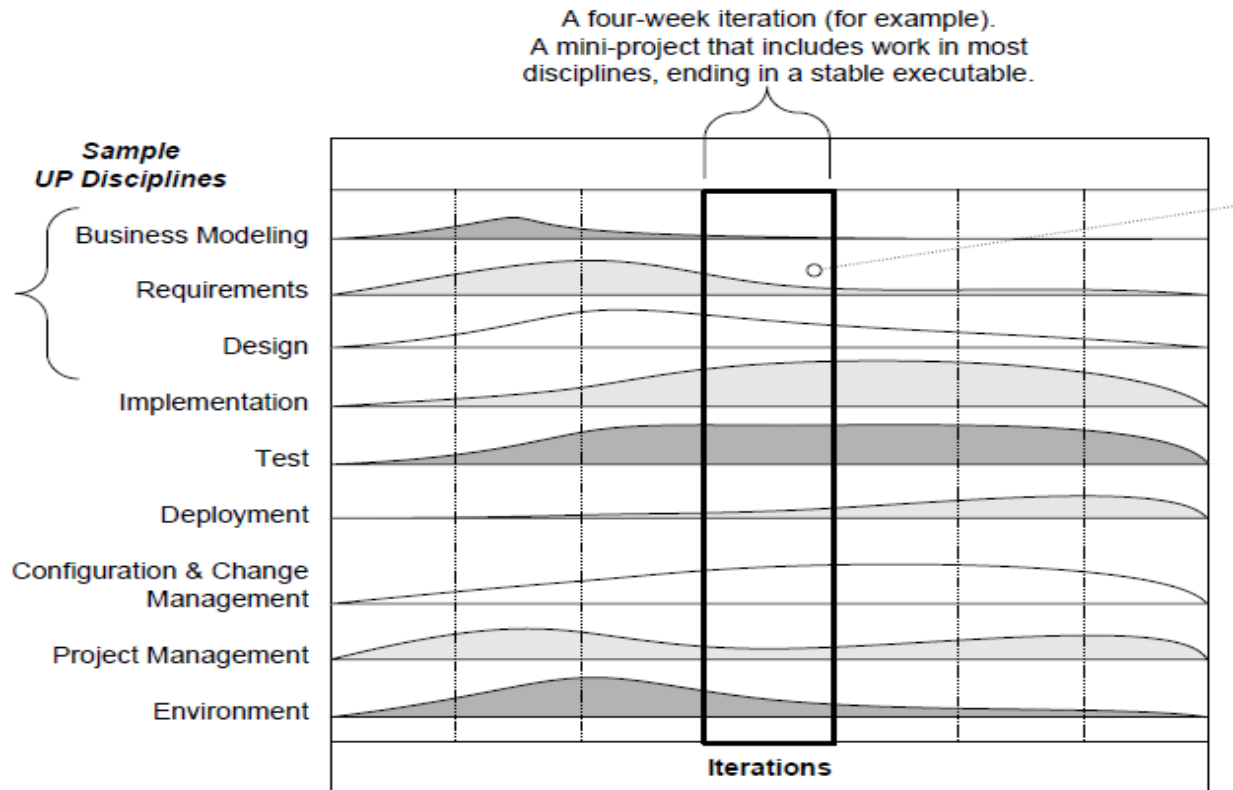


Fig: Sample UP Disciplines

UML DIAGRAMS

UML:

- Unified Modeling Language(UML) is a standard notation for the modeling of real-world objects as a first step in developing an object oriented design methodology.
- UML is a Visual language for specifying, constructing and documenting the artifacts of a system.
- The Various UML diagrams are as follows,
 - i. Use Case Diagram
 - ii. Class Diagram

- iii. Interaction Diagram
 - Sequence Diagram
 - Collaboration Diagram or Communication Diagram
- iv. State Diagram
- v. Activity Diagram
- vi. Package Diagram
- vii. Component Diagram
- viii. Deployment Diagram

Three ways to apply UML:

1. UML as sketch:

Informal and incomplete diagrams created to explore difficult parts of the problem.

2. UML as blueprint:

Detailed design diagram used for better understanding of code.

3. UML as programming language:

Complete executable specification of a software system in UML.

Three perspectives to apply UML:

1. Conceptual perspective: Diagrams describe the things of real world.

2. Specification perspective: Diagrams describe software abstractions or components with specifications and interfaces.

3. Implementation perspective: Diagrams describe software implementation in a particular technology.

USE CASE DIAGRAM

Use case diagrams are used to describe a set of actions (use cases) that some system or systems should or can perform in collaboration with one or more external users of the system (actors). Each use case should provide some observable and valuable result to the actors or other stakeholders of the system.


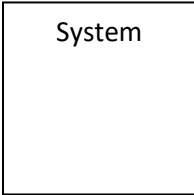

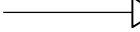
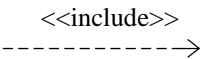
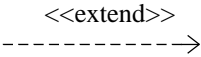
Purpose:

1. Used to gather requirements of a system
2. Used to get an outside view of a system
3. Identify external and internal factors influencing the system
4. Show the interaction among the requirements through actors.

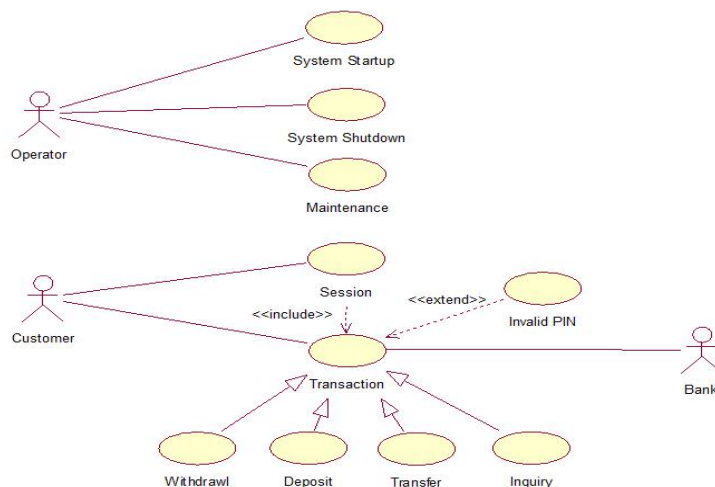
Uses:

1. Requirement analysis and high level design
2. Model the context of a system
3. Reverse engineering
4. Forward engineering

Notations:

S.No	Name	Notation	Description
1	Actor		Actors are the entities that interact with the system.
2	System		The use cases in the system make up the total requirements of the system.
3	Use Case		Use Case describes the actions performed by the user.
4	Generalization		A generalization relationship is used to represent inheritance relationship between model elements of same type.
5	Include		An include relationship specifies how the behavior for the inclusion use case is inserted into the behavior defined for the base use case.
6	Extend		An extend relationship specifies how the behavior of the extension use case can be inserted into the behavior defined for the base use case.

Sample Example - ATM System



CLASS DIAGRAM:

Class diagram is a static diagram. It represents the static view of an application. The class diagram describes the attributes and operations of a class and also the constraints imposed on the system. The class diagrams are widely used in the modeling of object oriented systems because they are the only UML diagrams which can be mapped directly with object oriented languages.

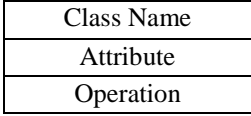
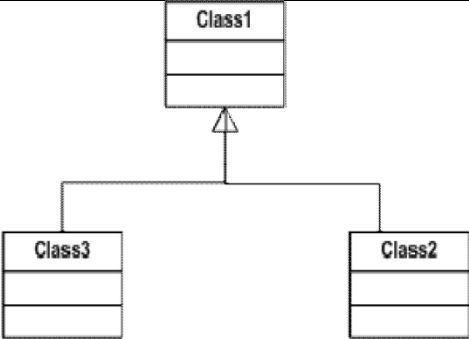
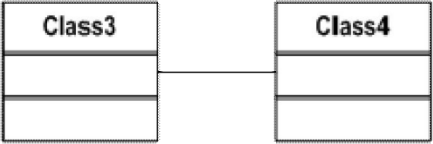
Purpose:

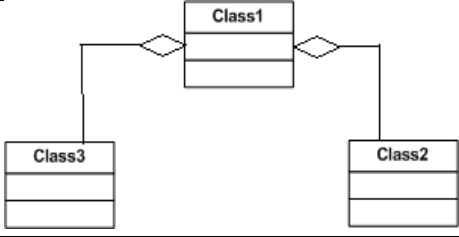

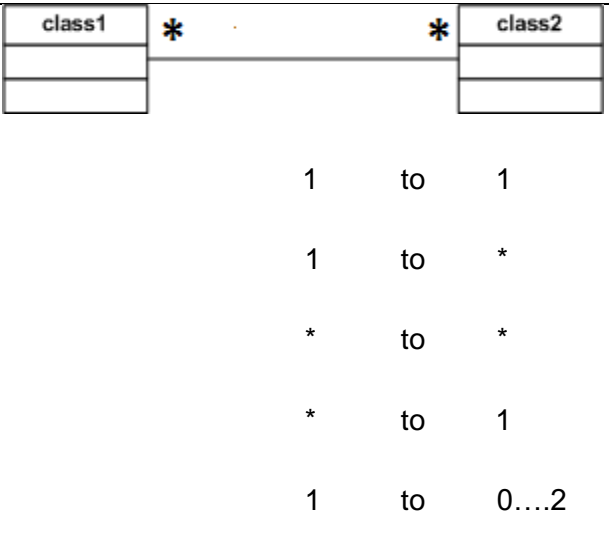
1. Analysis and design of the static view of an application
2. Describe responsibilities of a system
3. Base for Component and Deployment Diagrams
4. Forward and Reverse Engineering

Uses:

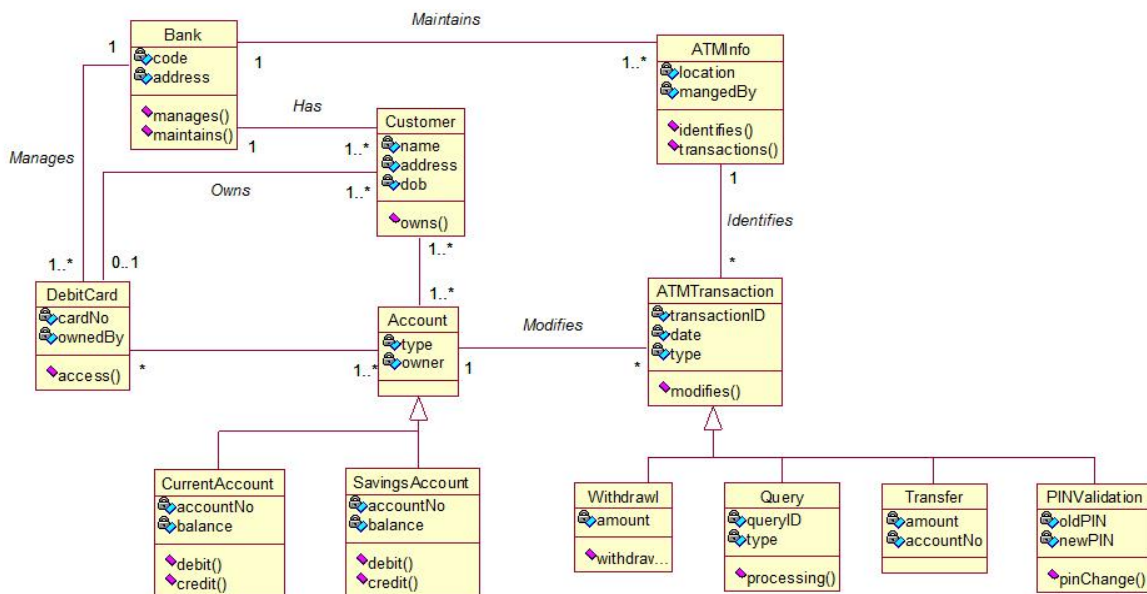
1. Describes the static view of the system
2. Shows the collaboration among the elements of the static view
3. Describes the functionalities performed by the system.
4. Construction of software applications using object oriented languages.

Notations:

S.No	Name	Notation	Description
1	Class		Class is an entity which describes a group of objects with same properties & behavior.
2	Generalization		Generalization refers to a relationship between two classes where one class is a specialized version of another.
3	Association		Association represent static relationships between classes.

4	Aggregation		Aggregation is a vague kind of association in the UML that loosely suggests whole-part relationships.
5	Composition		Composition is a strong kind of whole-part aggregation.
6	Multiplicity		Multiplicity specifies the number of instances of one class that may relate to a single instance of an associated class.

Sample Example – ATM System



INTERACTION DIAGRAM

Interaction diagrams are used to visualize the interactive behavior of the system. The Interactive behaviour is represented in UML by two diagrams namely,

- **Sequence Diagram**- It emphasizes on time sequence of messages
- **Collaboration Diagram**- It emphasizes on structural organization of the objects that send and receive messages.

Purpose:

1. To capture dynamic behaviour of a system
2. To describe the message flow in the system
3. To describe structural organization of the objects
4. To describe interaction among objects

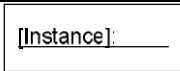
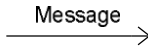
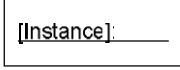
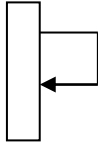
I. SEQUENCE DIAGRAM

Sequence diagram describes an interaction by focusing on the sequence of messages that are exchanged, along with their corresponding occurrence specifications on the lifelines.

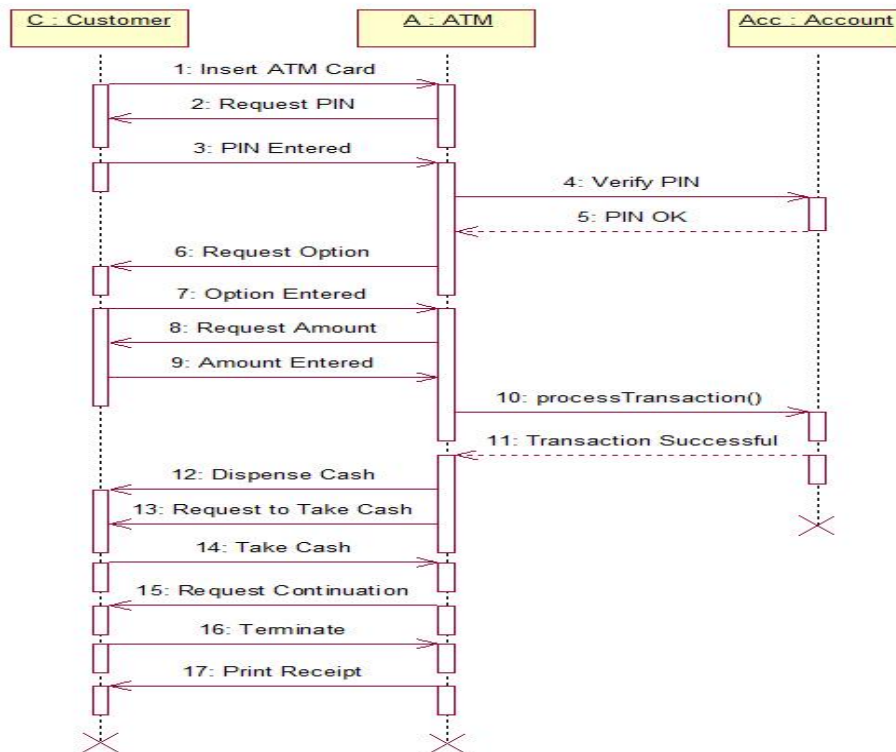
Uses:

1. To model flow of control by time sequence
2. To model flow of control by structural organizations
3. Forward engineering
4. Reverse engineering

Notations:

S.No	Name	Notation	Description
1	Lifeline	 A vertical dashed line extends downwards from the bottom center of the box.	Lifeline represents the duration during which an object is alive and interacting with other objects in the system.
2	Message		To send message from one object to another.
3	Object		It represents the existence of an object of a particular time.
4	Self message		Self message is a message by the object to itself.

Sample Example – ATM System



II. COLLABORATION DIAGRAM

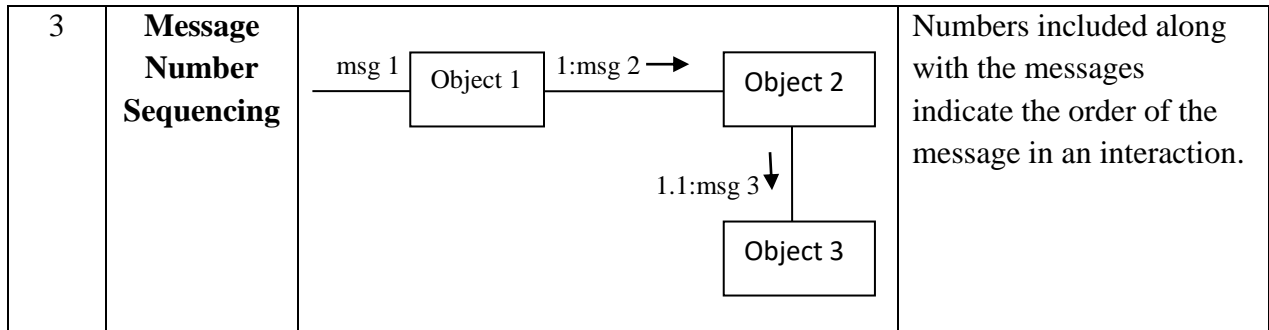
Collaboration or Communication diagram is also used to model the dynamic behaviour of the system. It emphasizes on structural organization of the objects that send and receive messages.

Uses:

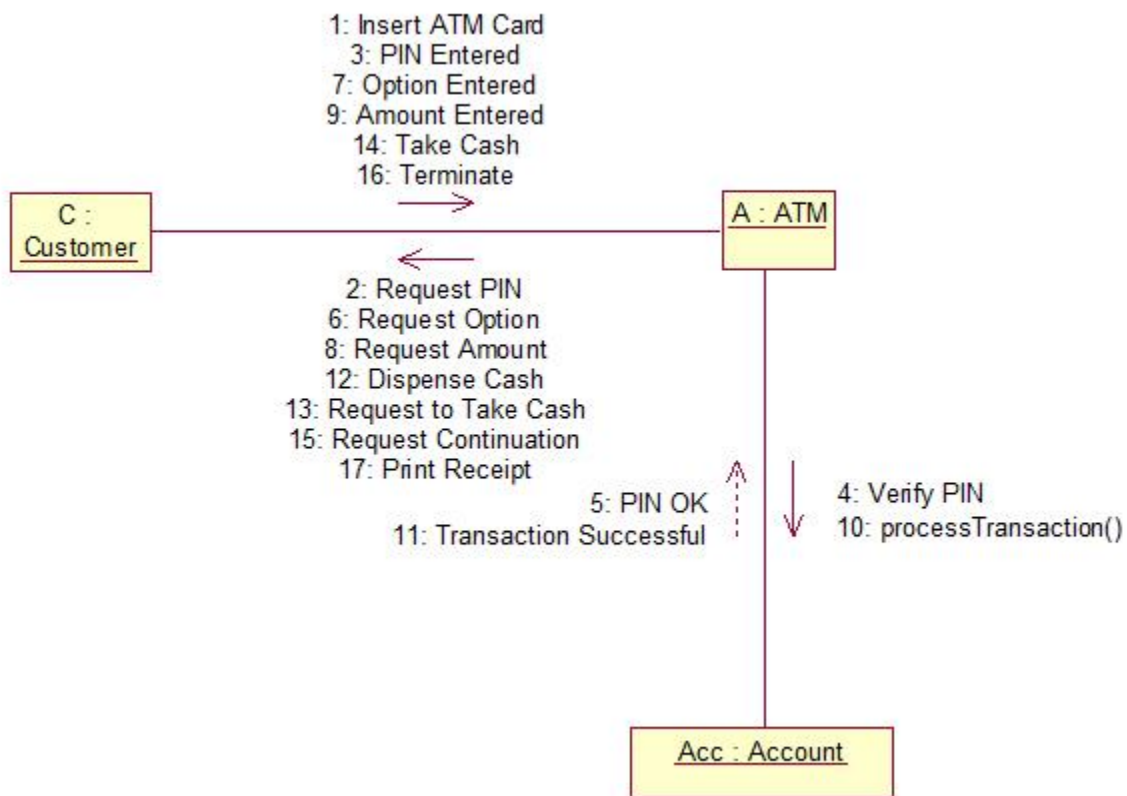
1. Used to show the messages that flow from one object to another within the system and the order in which they happen.
2. Used to track the source of the message from where it has been sent
3. Used to provide relationships and interactions among software objects

Notations:

S.No	Name	Notation	Description
1	Link	_____	A Link is a connection path between two objects
2	Message		Communication between objects takes place through messages. A sequence number is added to show the sequential order of messages.



Sample Example – ATM System



STATE DIAGRAM

- A State diagram is used to describe the behaviour of the systems. State diagrams require that the system described is composed of a finite number of states.
- State diagrams are used to give an abstract description of the behaviour of a system. This behaviour is analysed and represented in series of events, that could occur in one or more possible states.

Purpose:

1. It describes dynamic behavior of the objects of the system.
2. It specifies the possible states, what transitions are allowed between states.
3. It is used to describe the dependence of the functionality on the state of the system
4. The state model describes those aspects of objects concerned with time and the sequencing of operations events.

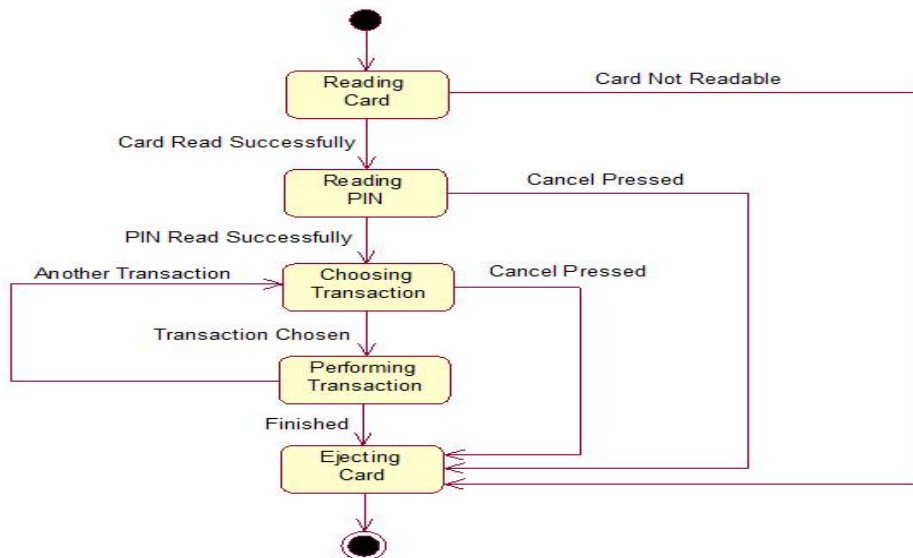
Uses:

1. To model the object states of a system.
2. To model the reactive system. Reactive system consists of reactive objects.
3. To identify the events responsible for state changes.
4. Forward and reverse engineering.

Notations:

S.No	Name	Notation	Description
1	Initial State	●	It shows the starting state of object.
2	Final State	⦿	It shows the terminating state of object.
3	State	▭	Represents the state of object at an instant of time
4	Transition	→	A transition is a directed relationship between a source state and a target state.

Sample Example – ATM System



ACTIVITY DIAGRAM

An Activity diagram is basically a flowchart to represent the flow from one activity to another activity. Activity diagrams are typically used for business process modeling, for modeling the logic captured by a single use case or usage scenario or for modeling the detailed logic of a business rule.

Purpose:

1. Draw the activity flow of a system
2. Describe the sequence from one activity to another
3. Describe the parallel, branched and concurrent flow of the system.





How to apply Activity Diagrams?

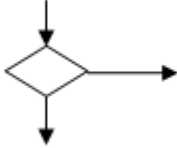
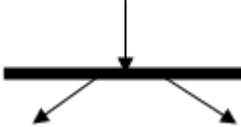
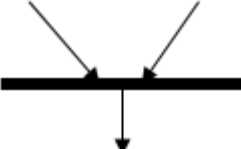
1. Activity diagrams show the flow of activities through the system.
2. Diagrams are read from top to bottom and have branches and forks to describe conditions and parallel activities.
3. A fork is used when multiple activities are occurring at the same time
4. The branch describes what activities will take place base on set of conditions
5. All branches at some point are followed by a merge to indicate the end of the conditional behavior started by that branch
6. After the merge all of the parallel activities must be combined by a join before transitioning into the final activity state.
7. Activity diagrams are applied to visualize business workflows and processes and use cases.

Uses:

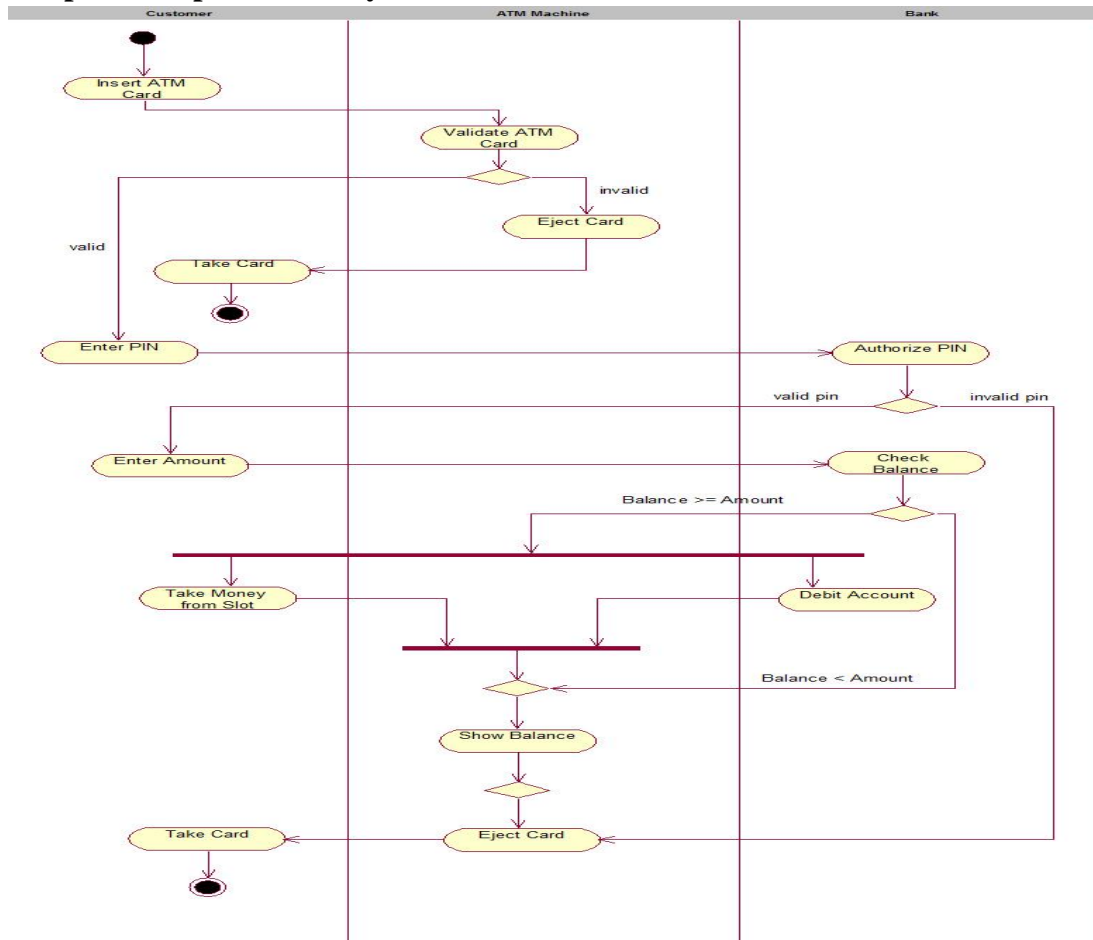
1. Visualize business processes and workflows.
2. Model work flow by using activities.
3. Model business requirements.
4. High level understanding of the system's functionalities.
5. Investigate business requirements at a later stage.

Notations:

S.No	Name	Notation	Description
1	Activity		Represents an individual activity of a system
2	Initial State		It shows the starting state of object.
3	Final State		It shows the terminating state of object.
4	Transition		Represents flow of data from one activity to another.

5	Decision		Decision node is a control node that accepts tokens on one or more incoming edges and selects outgoing edge from two or more outgoing flows.
6	Fork		A fork represents a single incoming transition and multiple outgoing transitions exhibiting parallel behavior
7	Join		A join in the activity diagram synchronizes the parallel behavior started at a fork.

Sample Example – ATM System



PACKAGE DIAGRAM

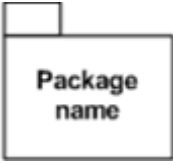
- Package diagrams organize the elements of a system into related groups to minimize dependencies among them.
- UML package diagrams are used to illustrate the logical architecture of a system, the layers, subsystems, packages etc.

Package is a namespace used to group together elements that are semantically related and might change together. It is a general purpose mechanism to organize elements into groups to provide better structure for system model.

Uses:

1. Package diagrams can use packages containing use cases to illustrate the functionality of a software system.
2. Package diagrams can use packages that represent the different layers of a software system to illustrate the layered architecture of a software system.

Notations:

S.No	Name	Notation	Description
1	Package		A package is a group of elements with common theme.

COMPONENT DIAGRAM

Component diagrams are used to model physical aspects of a system (elements like executables, libraries, files, documents etc.).Component diagrams are used to visualize the organization and relationships among the components in a system.


Purpose:

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

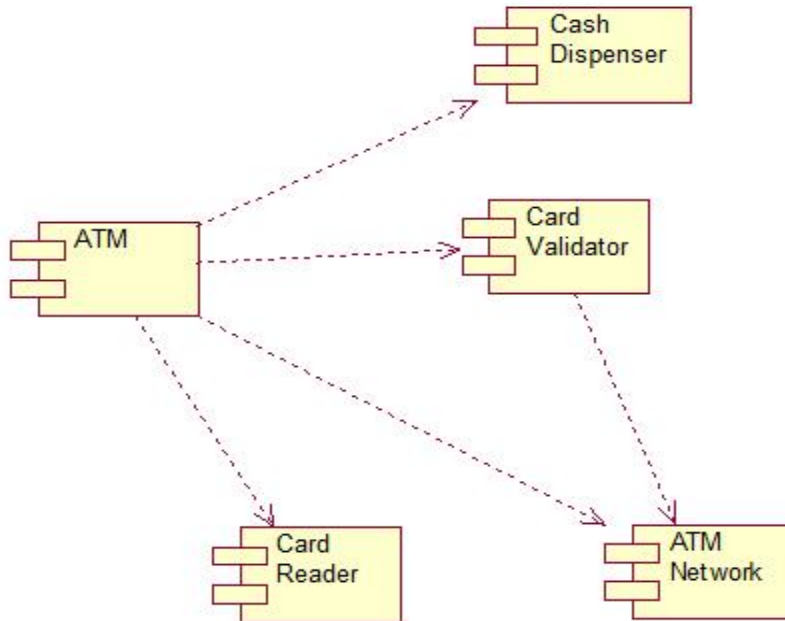
Uses:

1. Model the components of a system
2. Model database schema
3. Model executables of an application
4. Model system's source code.

Notations:

S.No	Name	Notation	Description
1	Component		A Component is a physical building block of the system

Sample Example – ATM System



DEPLOYMENT DIAGRAM

- Deployment diagram is defined as assignment of concrete software artifacts (executable files) to computational nodes (processing services).
- Deployment of software elements to the physical architecture and the communication (network) between physical elements.

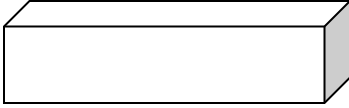
Purpose:

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

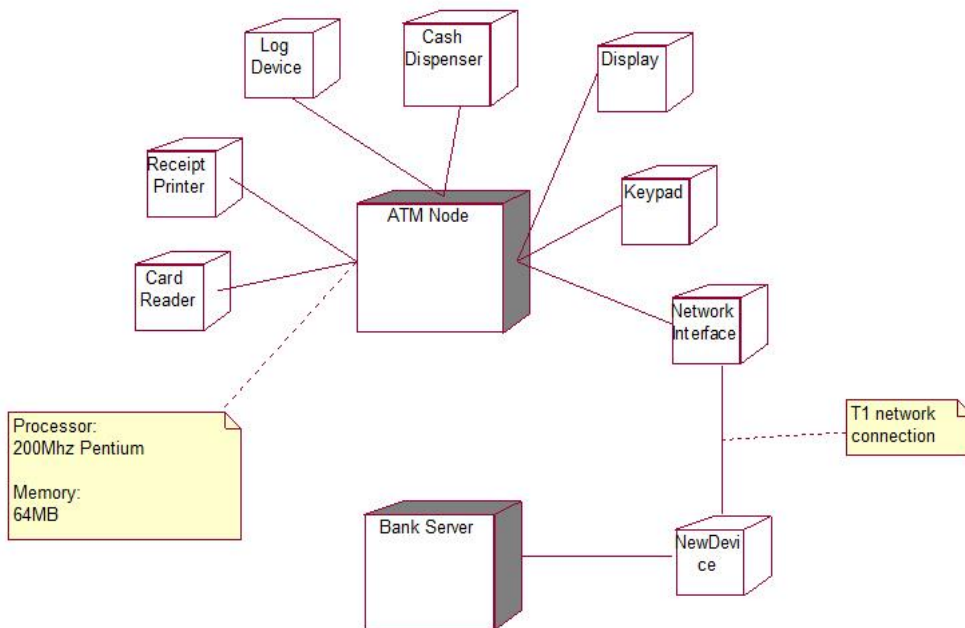
Uses:

1. To model the hardware topology of a system.
2. To model the embedded system.
3. To model the hardware details for a client/server system.
4. To model the hardware details of a distributed application.
5. Forward and Reverse engineering.

Notations:

S.No	Name	Notation	Description
1	Node		A single node in a deployment diagram represents multiple physical nodes, such as cluster of database servers.

Sample Example – ATM System



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.

GRASP: General Responsibility Assignment Software Patterns

- GRASP is a learning aid that helps to understand essential object design and apply design reasoning in a methodical, rational and explainable way.
- GRASP is used as a tool to help master the basics of OOD and understanding responsibility assignment in object design.
- There are nine basic OO design principles in GRASP. They are,
 1. Creator
 2. Information Expert
 3. Low Coupling
 4. High Cohesion
 5. Controller
 6. Polymorphism
 7. Pure Fabrication
 8. Indirection
 9. Protected Variations

CREATOR

Creation of objects is one of the most common activities in an object oriented system. Which class is responsible for creating objects is a fundamental property of relationship between objects of particular classes.

Problem

Who should be responsible for creating a new instance of some class?

Solution:

Assign class B the responsibility to create an instance of a class A if one of these is true,

- B contains or compositely aggregates A
- B records A
- B closely uses A
- B has the initializing data for A that will be passed to A when it is created.

Thus B is an expert with respect to creating A

B is a creator of A objects

If more than one option applies, usually prefer a class B which aggregates or contains A.

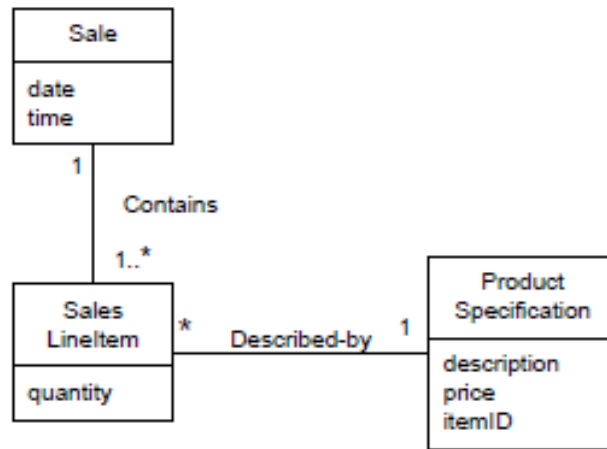


Fig: Partial Domain Model

Since a Sale contains many SalesLineItem objects, the Creator pattern suggests that Sale is a good candidate to have the responsibility of creating SalesLineItem instances.

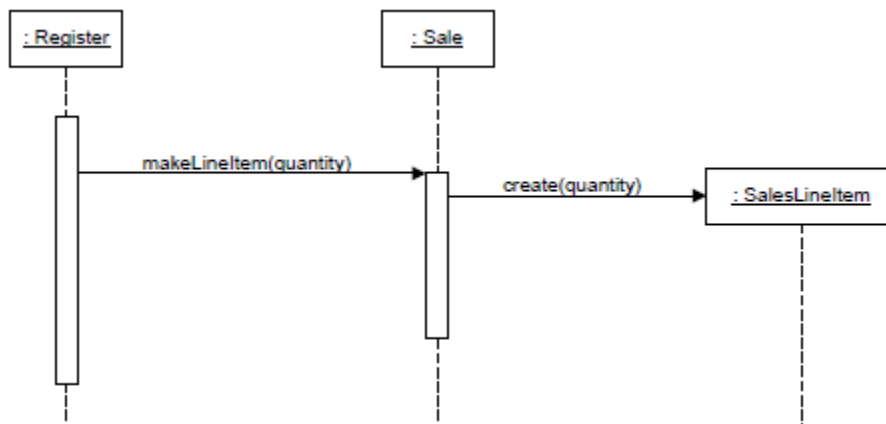


Fig: Creating a SalesLineItem

This assignment of responsibilities requires that a makeLineItem method be defined in Sale. The method section of class diagram can then summarize the responsibility assignment results, concretely realized as methods.

INFORMATION EXPERT

Problem

What is a general principle of assigning responsibilities to objects?

Solution:

Assign a responsibility to the information expert-the class that has the information necessary to fulfill the responsibility.

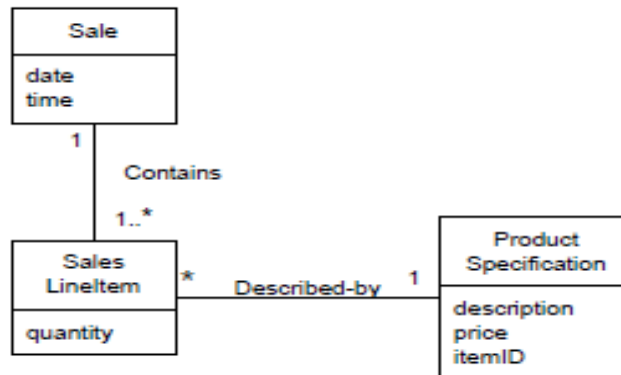


Fig: Partial domain model

- ❖ What information is needed to determine the grand total? A Sale instance contains these; therefore, by the guideline of Information Expert, Sale is a suitable class of object for this responsibility.
- ❖ The SalesLineItem knows its quantity and its associated ProductSpecification; therefore, by Expert, SalesLineItem should determine the subtotal; it is the information expert.

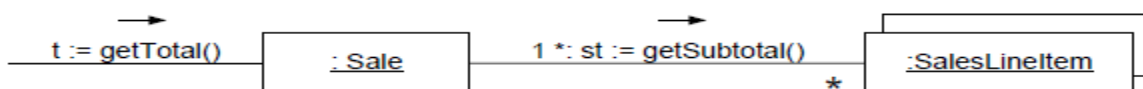


Fig: Partial interaction and class diagrams

- ❖ In terms of an interaction diagram, Sale needs to send get-Subtotal messages to each of the SalesLineItems and sum the results.

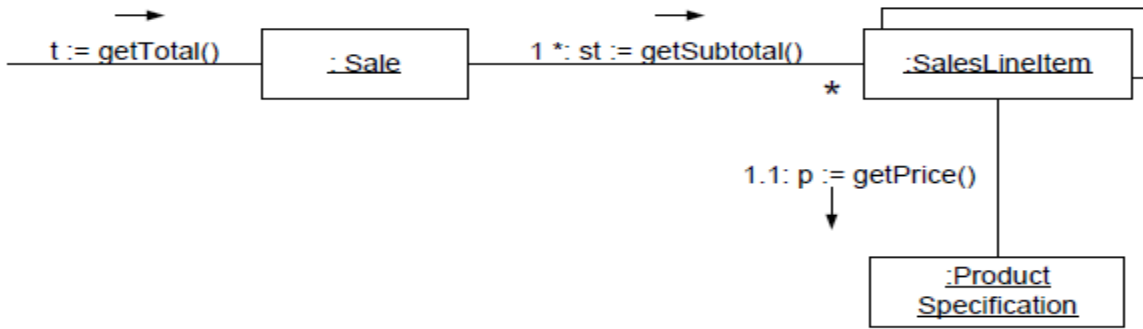


Fig: Calculating the Sale total

- ❖ The Product Specification is an Information Expert on answering its price, therefore SalesLineItem send it a message asking for the product price.

To fulfill the responsibility of knowing and answering the sale’s total, three responsibilities were assigned to three design classes of objects as follows.

Design Class	Responsibility
Sale	knows sale total
SalesLineItem	knows line item subtotal
ProductSpecification	knows product price

LOW COUPLING

Coupling is a measure of how strongly one element is connected to, has knowledge of, or relies on other elements. An element with low (or weak) coupling is not dependent on too many other elements.

A class with high (or strong) coupling relies on many other classes. Such classes may be undesirable; some suffer from the following problems,

- ❖ Forced local changes because of changes in related classes.
- ❖ Harder to understand in isolation.
- ❖ Harder to reuse because its use requires the additional presence of the classes on which it is dependent.

Problem

How to support low dependency, low change impact, and increased reuse?

Solution:

Assign a responsibility so that coupling remains low.

Eg: Partial Class domain



Assume that a Payment instance is to be created and associated with the Sale. What class should be responsible for this? Since a Register "records" a Payment in the real-world domain, the Creator pattern suggests Register as a candidate for creating the Payment. The Register instance could then send an addPayment message to the Sale, passing along the new Payment as a parameter.

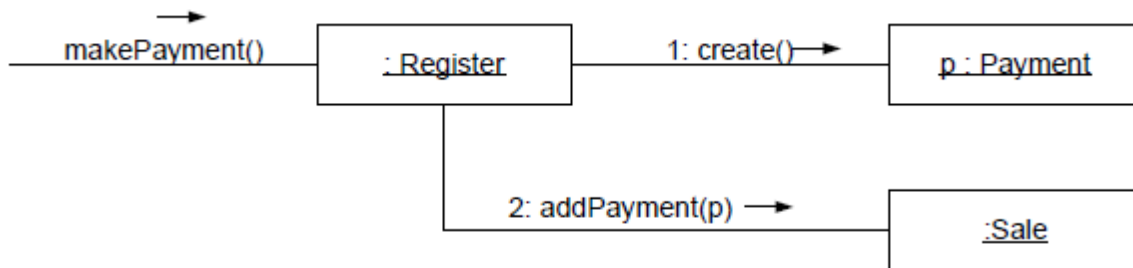


Fig: Register creates Payment

Assignment of responsibilities couples the Register class to knowledge of payment class. Alternative solution to create payment and associate it with Sale.

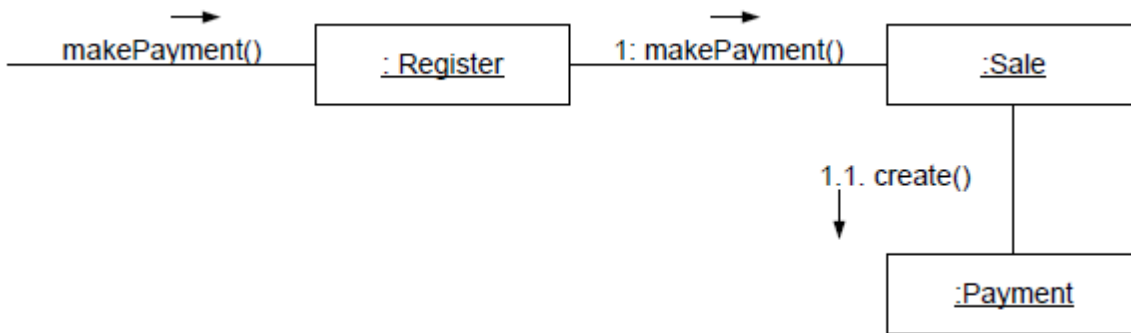


Fig: Sales creates Payment

In object-oriented languages such as C++, Java, and C#, common forms of coupling from TypeX to TypeY include:

- ✓ TypeX has an attribute (data member or instance variable) that refers to a TypeY instance, or TypeY itself.
- ✓ A TypeX object calls on services of a TypeY object.
- ✓ TypeX has a method that references an instance of TypeY, or TypeY itself, by any means. These typically include a parameter or local variable of type TypeY, or the object returned from a message being an instance of TypeY.
- ✓ TypeX is a direct or indirect subclass of TypeY.
- ✓ TypeY is an interface, and TypeX implements that interface.

HIGH COHESION

Cohesion

Cohesion is a measure of how strongly related and focused the responsibilities of an element are. An element with highly related responsibilities, and which does not do a tremendous amount of work, has high cohesion. These elements include classes, subsystems, and so on.

Problem

How to keep objects focused, understandable, and manageable, and as a side effect, support Low Coupling?

Solution:

Assign a responsibility so that cohesion remains high.

A class with low cohesion does many unrelated things, or does too much work. Such classes are undesirable; they suffer from the following problems:

- ✓ Hard to comprehend
- ✓ Hard to reuse
- ✓ Hard to maintain
- ✓ Delicate; constantly affected by change.

Example

Assume that a Payment instance is to be created and associate it with the Sale. What class should be responsible for this? Since Register records a Payment in the real-world domain, the Creator pattern suggests Register as a candidate for creating the Payment. The Register instance could then send an addPayment message to the Sale, passing along the new Payment as a parameter.

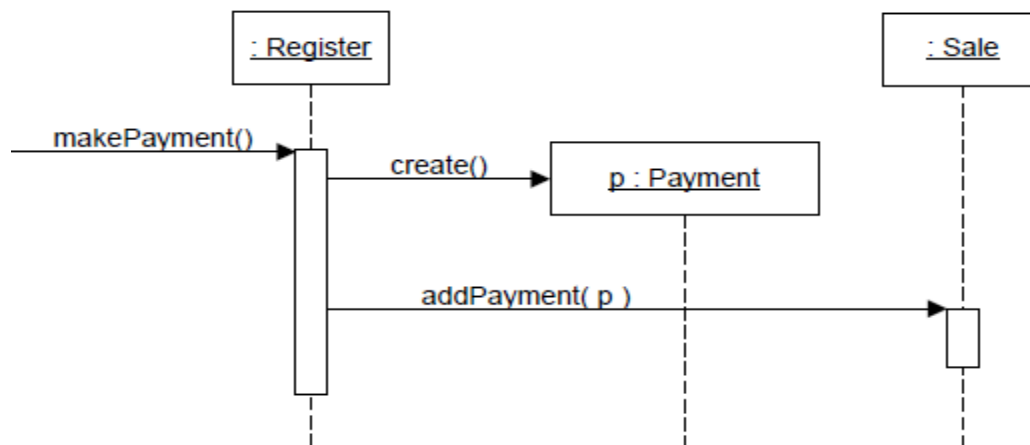


Fig: Register creates payment

This assignment of responsibilities places the responsibility for making a payment in the Register.

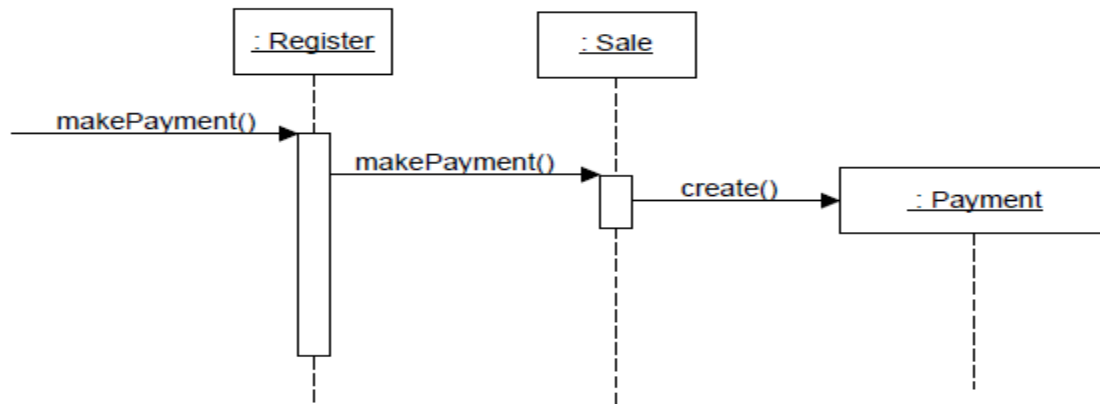


Fig: Sale creates Payment

CONTROLLER

A Controller is the first object beyond the UI layer that is responsible for receiving or handling a system operation message.

Problem

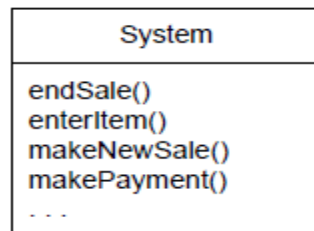
What first object beyond the UI layer receives and coordinates(controls) a system operation?

Solution:

Assign the responsibility to a class representing one of the following choices,

- ✓ Represents the overall system, “a root object”, a device that the software is running within, or a major subsystem.
- ✓ Represents a use case scenario within which the system event occurs.

Example: NextGen POS application



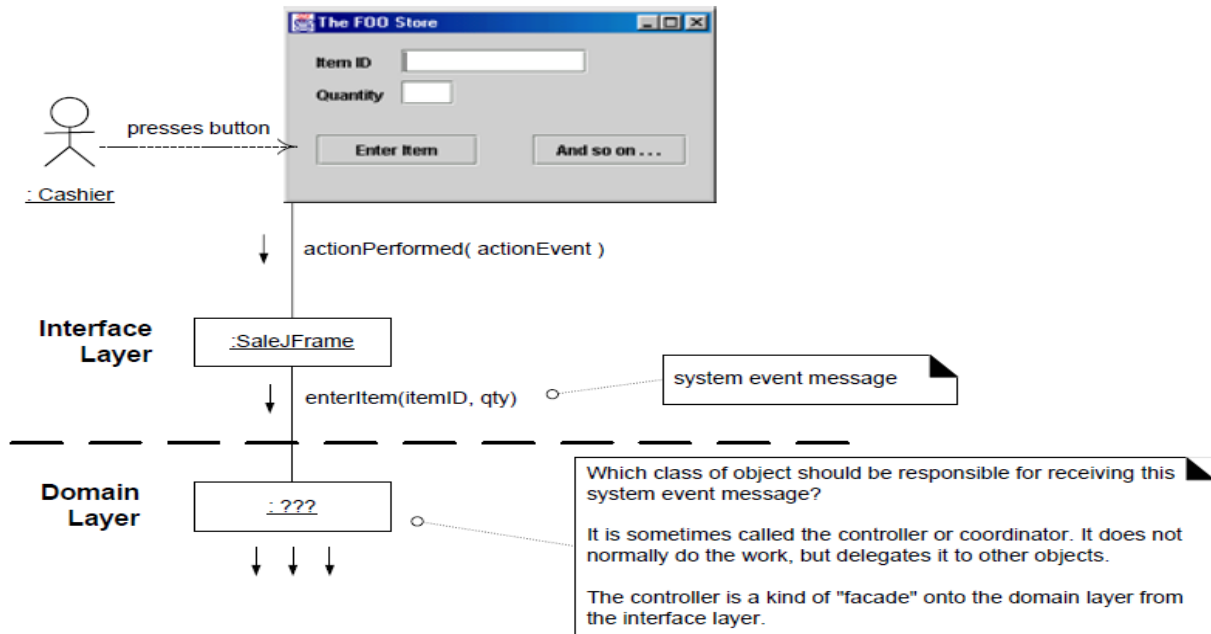


Fig: Assigning responsibilities to controller class

During design, a controller class is assigned the responsibility for system operation.

The system Operations identified during system behaviour analysis are assigned to one or more controller classes, such as Register,

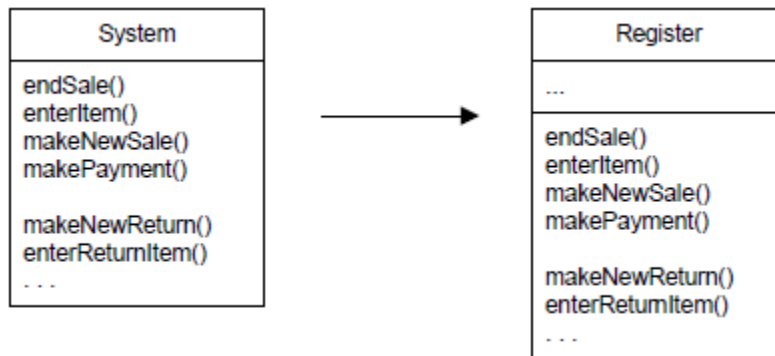


Fig: Controller Class

Bloated Controller

Poorly designed, a controller class will have low cohesion. unfocused and handling too many areas of responsibility; this is called a bloated controller.

Signs of bloating include:

- ✓ There is only a single controller class receiving all system events in the system, and there are many of them.

- ✓ The controller itself performs many of the tasks necessary to fulfill the system event, without delegating the work
- ✓ A controller has many attributes, and maintains significant information about the system or domain, which should have been distributed to other objects, or duplicates information found elsewhere.

Cures for a bloated controller

- ✓ Add more controllers-a system does not have to have only one. For example, consider an application with many system events, such as an airline reservation system.

Use-case controllers
MakeReservationHandler
ManageSchedulesHandler
ManageFaresHandler

- ✓ Design the controller so that it primarily delegates the fulfillment of each system operation responsibility on to other objects.

DESIGN PATTERNS

- Design patterns are termed as reusable solutions for commonly occurring problems in software designs.
- Design Patterns are descriptions of communicating objects and classes customized to solve a general design problem in a particular context.
- Design Patterns identifies the participating classes and instances, their roles and collaborations, and the distribution of responsibilities.

Essential Elements of a pattern:

1. Pattern Name
2. Problem
3. Solution
4. Consequences

Design Patterns are Categorized into,

1. Creational Pattern
 - Factory Method
2. Structural Patterns
 - Bridge
 - Adapter
3. Behavioral Patterns
 - Strategy
 - Observer

CREATIONAL PATTERN

Creational design patterns provide a way to create objects while hiding the creation logic, rather than instantiating objects directly using new operator. This gives program more flexibility in deciding which objects need to be created for a given use case.

FACTORY METHOD

Name: Factory

Problem: Who should be responsible for creating objects when there are special considerations, such as complex creation logic, a desire to separate the creation responsibilities for better cohesion, and so forth?

Solution: Create a Pure Fabrication object called a Factory that handles the creation.

Advantages of Factory objects

- ✓ Separate the responsibility of complex creation into cohesive helper objects.
- ✓ Hide potentially complex creation logic.
- ✓ Allow introduction of performance-enhancing memory management strategies, such as object caching or recycling.

In the below diagram, In ServicesFactory, the logic to decide which class to create is resolved by reading in class name from an external source and then dynamically loading the class. This is termed as **Partial Data Driven Design**.

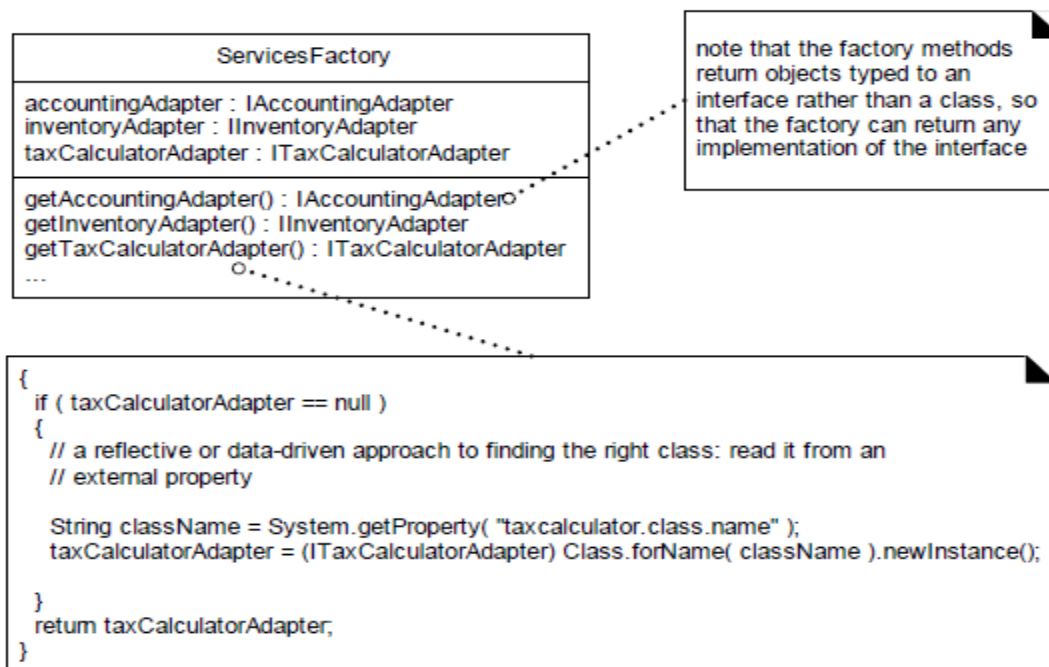


Fig: Factory Pattern

Benefits of Factory Method

- ✓ Factory method introduces a separation between the application and a family of classes. It provides a simple way of extending the family of products with minor changes in the application code
- ✓ It provides customization hooks. When the objects are created directly inside the class, it is hard to replace them by objects which extend their functionality. If a factory is used instead to create a family of objects that customizes objects can easily replace the original objects, configuring the factory to create them.

Drawbacks of Factory Method

- ✓ The Factory has to be used for a family of objects. If the classes doesn't extend common base class or interface they cannot be used in a factory design template.

Uses:

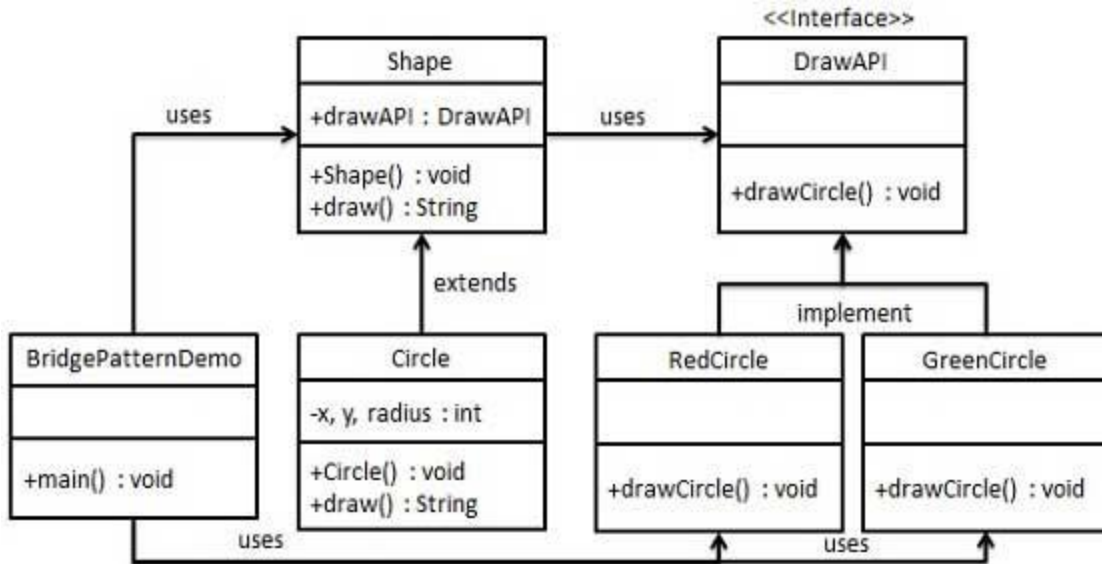
- ✓ Factory is used to manipulate objects of same type as abstract objects.
- ✓ Whenever an application is designed, factory plays a vital role in creating objects.

STRUCTURAL PATTERNS

Structural patterns are concerned with how classes and objects are composed to form larger structures. These patterns describe ways to compose objects to realize new functionality.

BRIDGE PATTERN

- Bridge is used, when we need to decouple an abstraction from its implementation so that the two can vary independently. This type of design pattern comes under structural pattern as this pattern decouples implementation class and abstract class by providing a bridge structure between them.
- This pattern involves an interface which acts as a bridge which makes the functionality of concrete classes independent from interface implementer classes. Both types of classes can be altered structurally without affecting each other.
- In the following diagram, DrawAPI interface is acting as a bridge implementer and concrete classes RedCircle, GreenCircle implementing the DrawAPI interface. Shape is an abstract class and will use object of DrawAPI. BridgePatternDemo, a demo class will use Shape class to draw different colored circle.



ADAPTER PATTERN

Adapter pattern works as a bridge between two incompatible interfaces. It is used to convert the programming interface of one class into that of another.

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.

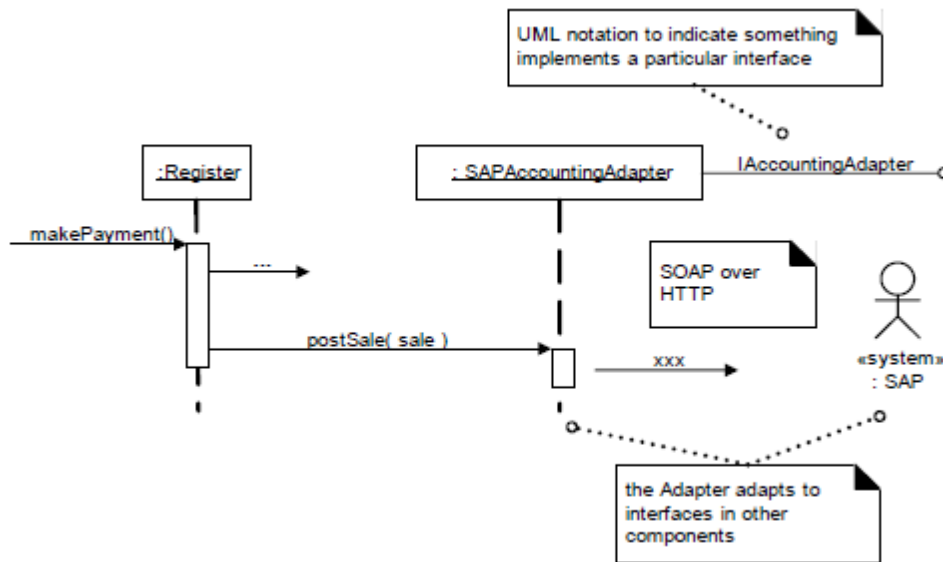
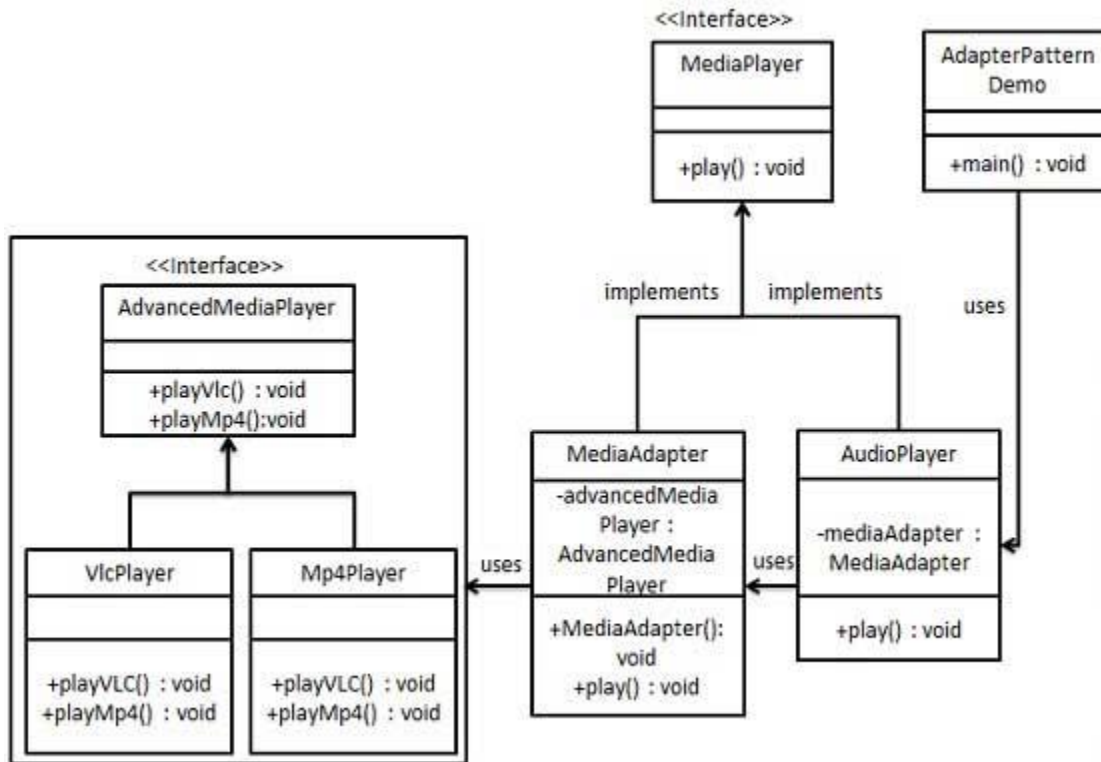


Fig: Adapter Pattern

Example:

Consider following example in which an audio player device can play mp3 files only and wants to use an advanced audio player capable of playing vlc and mp4 files with the use of adapter.



- In this example, we have a **MediaPlayer** interface and a concrete class **AudioPlayer** implementing the **MediaPlayer** interface. **AudioPlayer** can play mp3 format audio files by default.
- We are having another interface **AdvancedMediaPlayer** and concrete classes implementing the **AdvancedMediaPlayer** interface. These classes can play vlc and mp4 format files.
- We want to make **AudioPlayer** to play other formats as well. To attain this, we have created an adapter class **MediaAdapter** which implements the **MediaPlayer** interface and uses **AdvancedMediaPlayer** objects to play the required format.
- **AudioPlayer** uses the adapter class **MediaAdapter** passing it to the desired audio type without knowing the actual class which can play the desired format. **AdapterPatternDemo**, a demo class will use **AudioPlayer** class to play various formats.

BEHAVIORAL PATTERNS

Behavioral Patterns are concerned with communication between objects. These patterns use inheritance to distribute behavior between classes.

STRATEGY PATTERN

Problem

How to design for varying, but related, algorithms or policies? How to design for the ability to change these algorithms or policies?

Solution:

Define each algorithm/policy/strategy in a separate class, with a common interface.

Example:

In this example we are going to create a Strategy interface defining an action and concrete strategy classes implementing the Strategy interface. Context is a class which uses a Strategy.

StrategyPatternDemo, a demo class, will use Context and strategy objects to demonstrate change in Context behaviour based on strategy it deploys or uses.

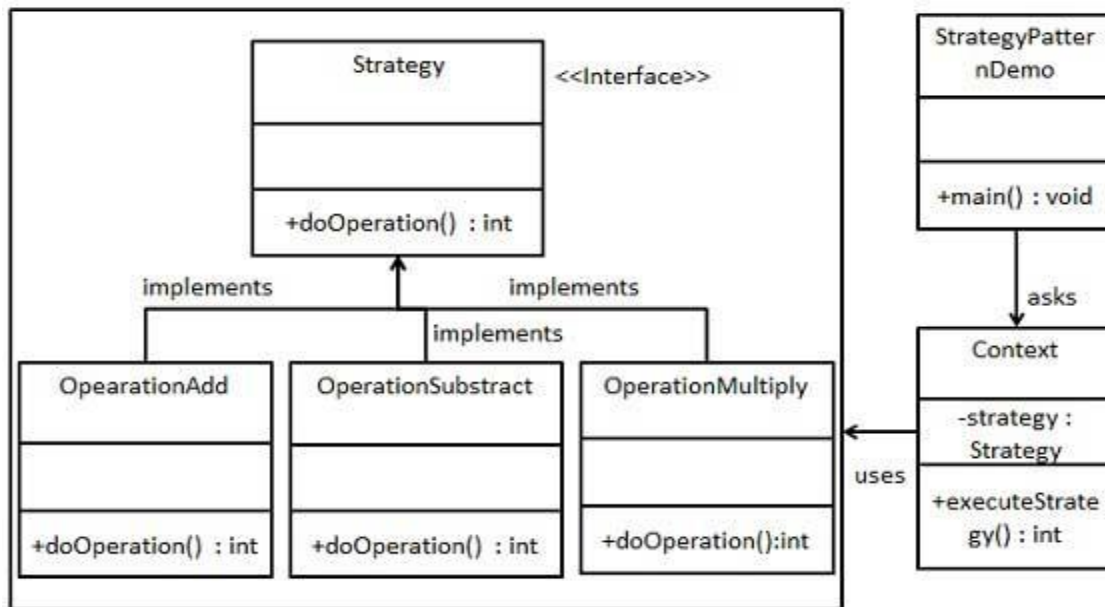


Fig: Strategy Pattern

OBSERVER PATTERN(Publish-Subscribe)

Problem

Different kinds of subscriber objects are interested in state changes or events of a publisher object, and want to react in their own unique way when the publisher generates an event. Moreover, the publisher wants to maintain low coupling to the subscribers. What to do?

Solution:

Define a “subscriber” or “listener” interface. Subscribers implement this interface. The publisher can dynamically register subscribers who are interested in an event and notify them when an event occurs.

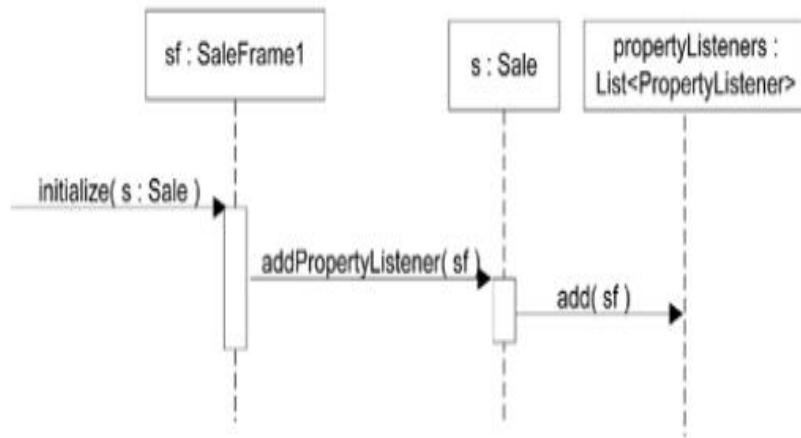


Fig: The observer SaleFrame1 subscribes to the publisher Sale

The SaleFrame1 object is the observer/subscriber/listener. In the above diagram it subscribes to interest in property events of the Sale, which is a publisher of property events. The Sale adds the object to its list of PropertyListener subscribers. The Sale does not know about the SaleFrame1 as a SaleFrame1 object, but only as a PropertyListener object, this lowers the coupling from the model up to the view layer.

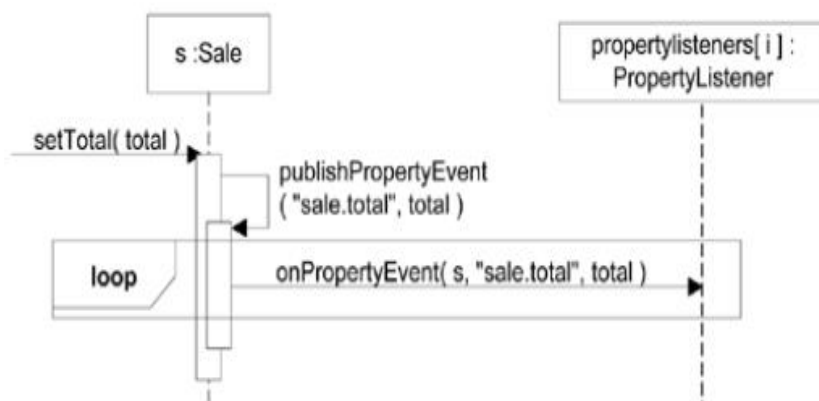


Fig: The Sale publishes a property event to all its subscribers

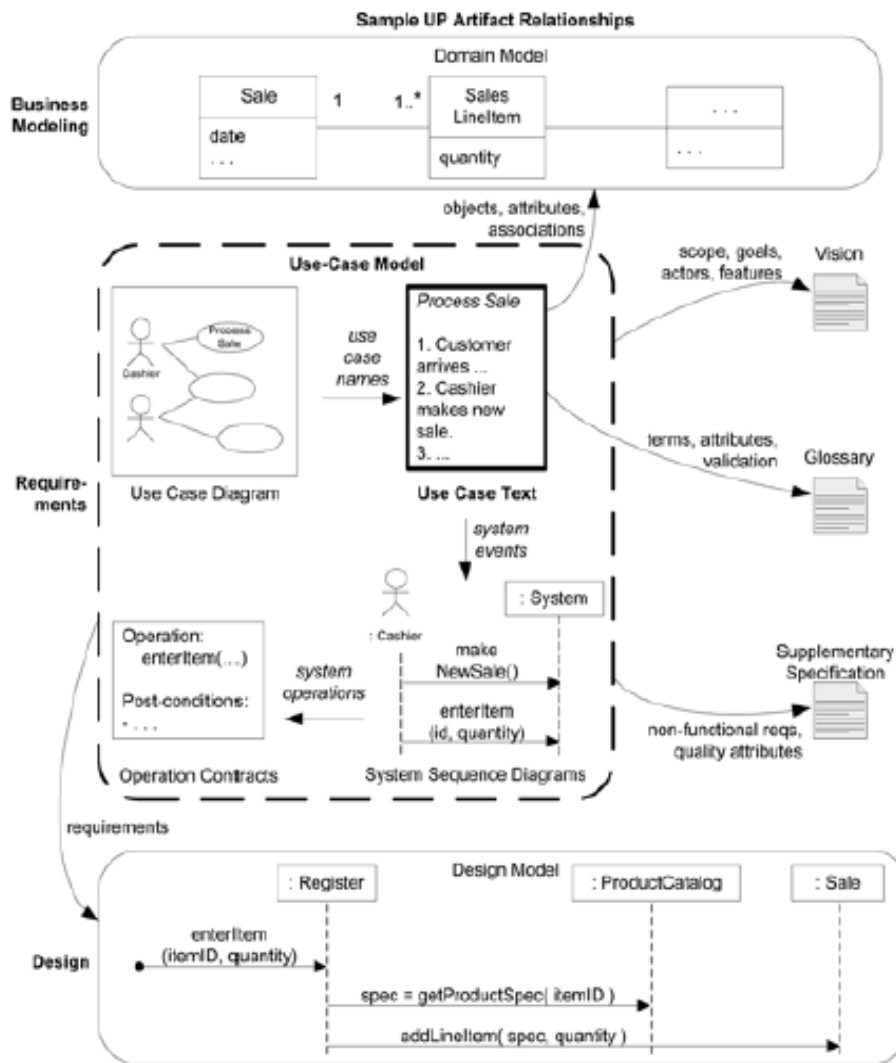
In the above diagram, when the sale total changes, it iterates across all its registered subscribers, and “publishes an event” by sending the onPropertyEvent message to each.

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.

CASE STUDY- The Next Gen POS system



- Next Generation Point of Sale (POS) system is a computerized application to record sales and handle payments.

- It is used in a retail store
- It includes hardware components such as a computer and bar code scanner and software to run the system
- It can be interfaced with the various service applications such as calculator and inventory control.
- POS is a fault tolerant system i.e if any of remote service fails, other services can be utilized
- POS system supports multiple and varied client-side terminals and interfaces. It includes
 - Thin-client web browser terminal
 - Regular personal computer with Java swing GUI
 - Touch screen input
 - Wireless PDAs etc.

INCEPTION

Inception is the initial stage of the project. Inception is not a requirements phase but it is a feasibility phase where complete investigation takes place to support a decision to continue or stop .It deals with

- Approximate vision
- Business case
- Scope
- Vague estimates

USE CASE MODELING

Use case model provides an external view of the system or application directed towards the users or the actors of the system. Use case model expresses what the business or application will do.

Use case Diagram

A use case diagram is a graph of actors, a set of use cases enclosed by a system boundary, communication associations between the actors and the use cases and generalization among the use cases.

Actors

An actor is an entity that interacts with a use case (object, place, or person)

Eg:Cashier

Scenario

A Scenario is a specific sequence of actions and interactions between actors and the system. It is also called as **use case instance**.

Use cases

- A use case is a static description of some way in which a system or a business is used by its users or actors.
- Use case is a collection of related success and failure scenarios that describe an actor using a system to achieve the goal.

Use cases and use case model

- Use cases are defined as text documents not as diagrams in unified process(UP).
- Use case model in UP is an act of writing text not drawing diagrams.
- Use case model in UP optionally includes UML use case diagram and it also consists of
 - Vision
 - Glossary
 - Business Rules
 - Supplementary Specification

Three Kinds of Actors

1. **Primary Actor** (to find user goals)- This kind of Actor satisfies the user goals through SUD (System Under Discussion) services. Eg.Librarian
2. **Supporting Actor** (to provide clear picture of external interfaces and protocols)- These actors provide a service information to SUD. Eg. Library assistant or computer system providing library details (Book or Transaction Details)
3. **Offstage Actor** (to ensure all the goals are identified and satisfied).

USE CASE FORMATS

Use case can be written in one of the following formats,

1. **Brief**- use case is a one paragraph summary consists of main success scenario.
2. **Casual**-use case is an informal paragraph i.e multiple paragraphs with various scenarios.
3. **Fully dressed**- Use cases are written in detail with supporting sections such as pre conditions and success guarantees.

RELATING USE CASES

Use cases can be related to each other using ,

1. Include
2. Extend
3. Generalization

INCLUDE

Include is a directed relationship between two use cases, implying that the behavior of the included use case is inserted into the behavior of including use case.

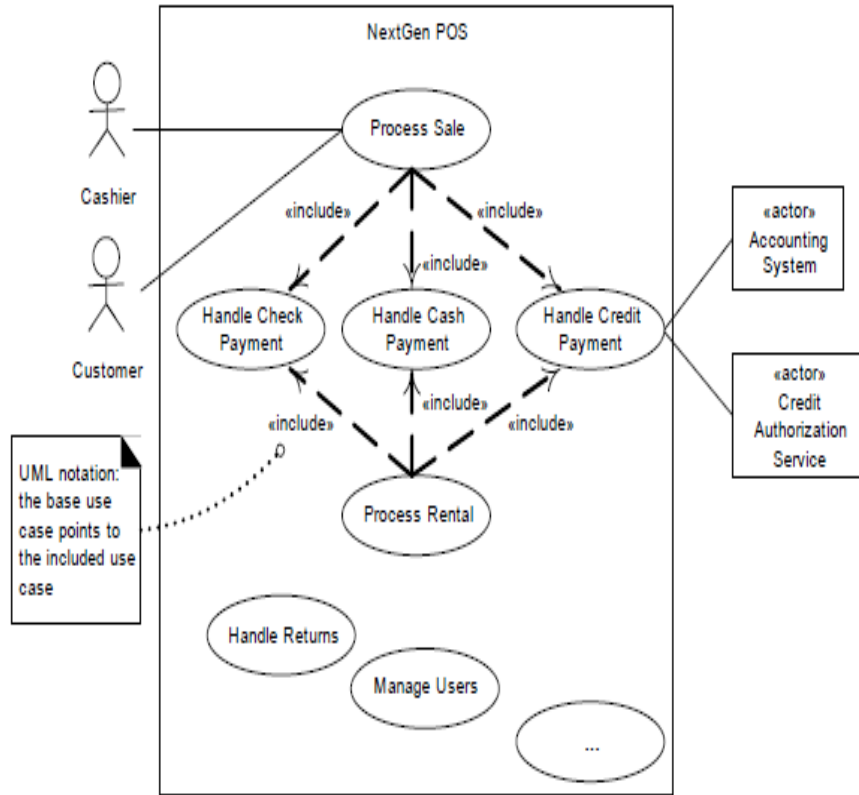


Fig: Use case include relationship in the Use-Case Model

EXTEND

An extend relationship specifies that one use case (extension) extends the behavior of another use case (base use case).

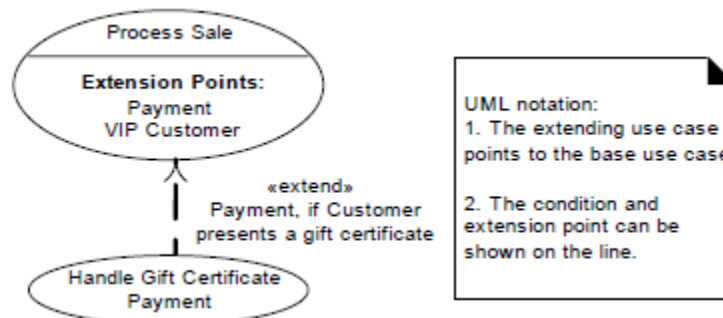


Fig: The extend relationship

GENERALIZATION

Generalization is the activity of identifying commonality among concepts and defining superclass (general concept) and subclass (specialized concept) relationships.

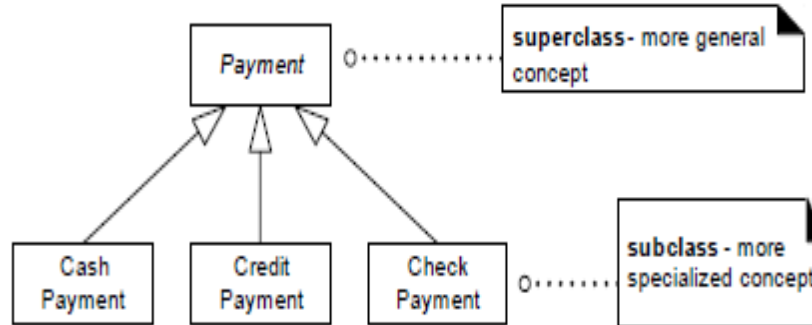


Fig: Generalization-specialization hierarchy

ELABORATION

Elaboration is the initial series of iterations during which the team performs the following actions,

1. Investigation
2. Implements the core architecture
3. Clarifies most requirements
4. Tackles high risk issues.

DOMAIN MODELS

- A domain model is a visual representation of conceptual classes or real-world objects in a domain. They are called **conceptual models**, **domain object models**, and **analysis object models**.
- Domain model can be represented by a set of class diagrams in which no operations (methods) are defined. It provides a conceptual view that includes,
 1. Domain objects or conceptual classes
 2. Association between conceptual classes
 3. Attributes of conceptual classes

DOMAIN MODEL AS A VISUAL DICTIONARY

- Domain model provides a visualization of concepts or words in Business domain such as name of the classes, association and attributes using UML notation.
- The information expressed by the Domain model can also be expressed by a plain text as a glossary and hence the name Domain model a visual dictionary.

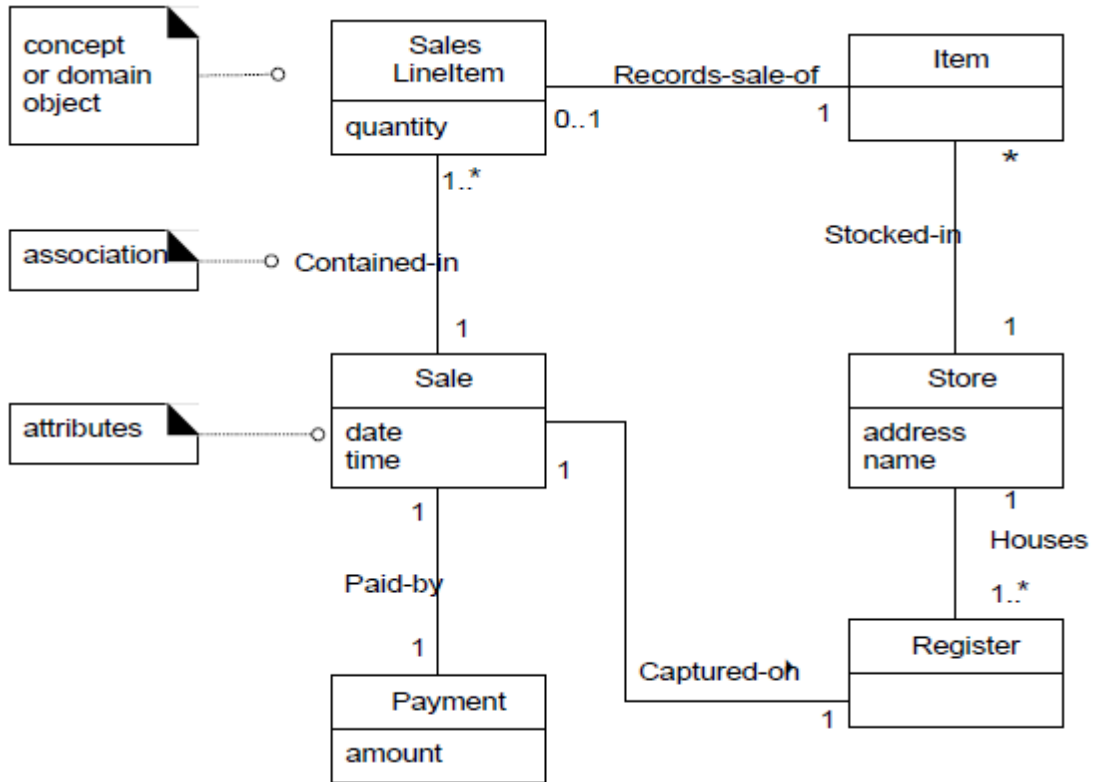


Fig: Partial domain model-a visual dictionary

CONCEPTUAL CLASSES

A Conceptual class is an idea, thing, or object to understand the real world situation. It is considered in terms of its symbol, intension, and extension.

- **Symbol**-words or images representing a conceptual class.



Fig: A conceptual class has a symbol

- **Intension**-the definition of a conceptual class.

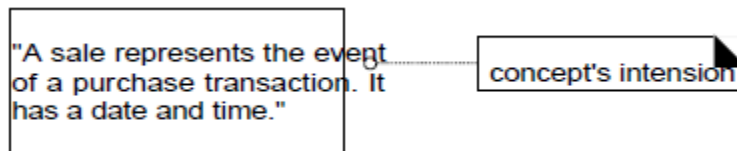


Fig: A conceptual class has an intension

- **Extension**-the set of examples to which the conceptual class applies.



Fig: A conceptual class has an extension

GUIDELINES TO CREATE A DOMAIN MODEL

1. Find the conceptual classes
2. Draw them as classes in a UML class diagram
3. Add associations and attributes

1) STRATEGIES TO FIND THE CONCEPTUAL CLASS

1. Reuse or modify existing models
 - They are published, well-crafted domain models and data models for many common domains such as inventory, finance, health etc.
2. Use of category list

S.No	Conceptual Class Category	Examples
1	Business Transactions	Sale,Payment
2	Roles of people	Cashier,Customer
3	Catalogs	Product catalog, Flight catalog
4	Records of Finance	Receipt,ledger

3. Identify noun phrases
 - Linguistic analysis i.e identify noun and noun phrases in textual description of a domain.
 - **Eg.** POS domain

A list of candidate classes for the domain is generated.

Sale	Cashier
CashPayment	Customer
SalesLineItem	Store
Item	ProductDescription

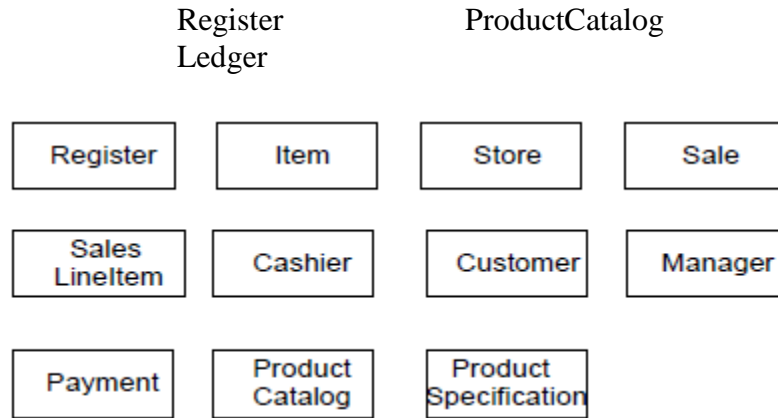
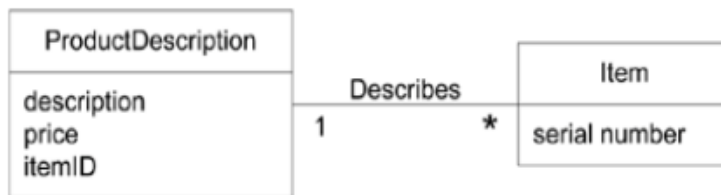


Fig: Initial POS domain model

DESCRIPTION CLASS

A description class contains information that describes something else.

Eg: ProductDescription- records price, picture and text description of an item.



USE OF A DESCRIPTION CLASS

Description class is used when,

- There needs to be a description about an item or service, independent of the current existence of any examples of those items or services.
- Deleting instances of things they describe results in a loss of information that needs to be maintained.
- It reduces redundant or duplicated information.

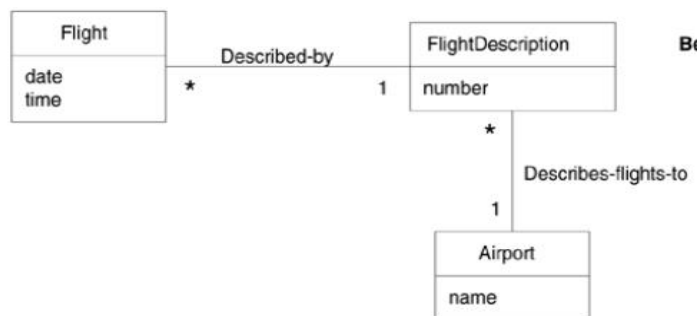


Fig: Description Class

ASSOCIATION

An association is a relationship between classes that indicates some meaningful and interesting connection.

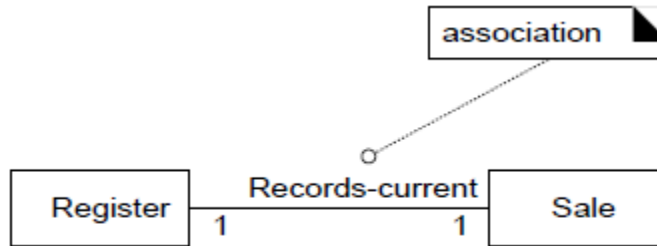


Fig: Association

ASSOCIATION NOTATION

- An association is represented as a line between classes with a capitalized association name.
- The end of an association contains a multiplicity expression indicating the numerical relationship between instances of the classes.

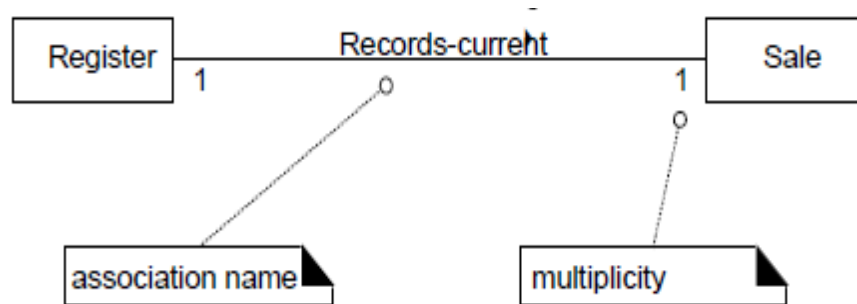


Fig: UML notation for association

MULTIPLICITY

Multiplicity defines how many instances of a class A can be associated with one instance of a class B.

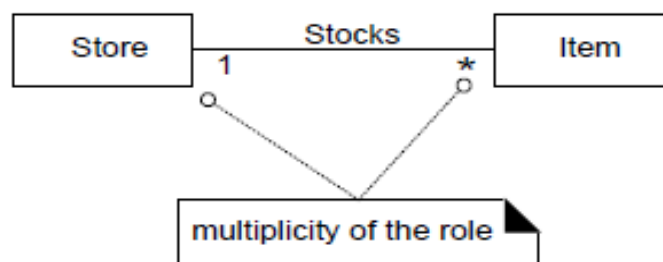


Fig: Multiplicity on an association

MULTIPLE ASSOCIATION BETWEEN TWO CLASSES



Fig: Multiple associations

ATTRIBUTE

- An attribute is a logical data value of an object.
- It is useful to identify those attributes of conceptual classes that are needed to satisfy the information requirements of the current scenarios under development.

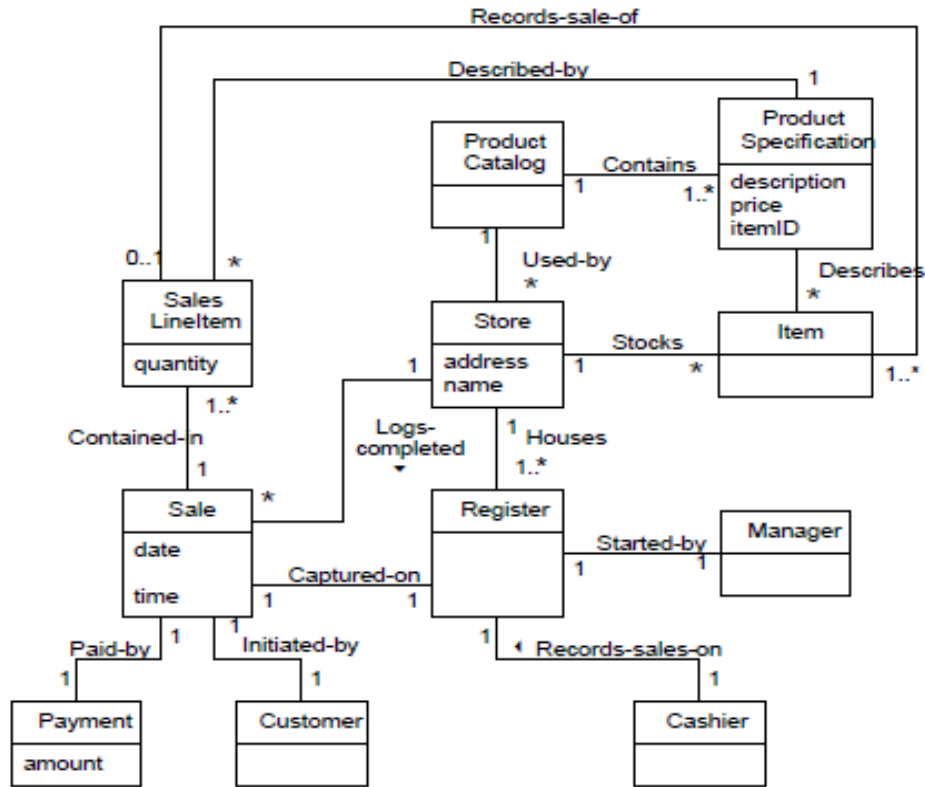


Fig: Partial Domain Model

DOMAIN MODEL REFINEMENT

Refinement of Domain Model is done with,

- Generalization
- Specialization
- Conceptual Class Hierarchies

GENERALIZATION

Generalization is the activity of identifying commonality among concepts and defining superclass (general concept) and subclass (specialized concept) relationships.

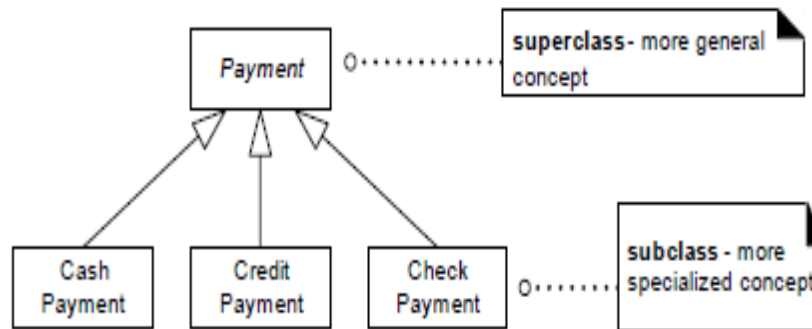


Fig: Generalization-specialization hierarchy

GENERALIZATION AND CLASS SETS

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

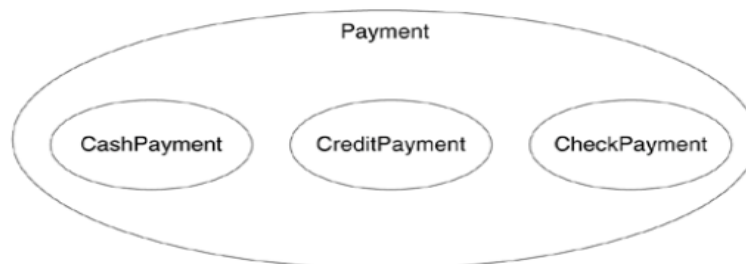


Fig: Venn diagram of set relationships

CONCEPTUAL SUBCLASS DEFINITION CONFORMANCE

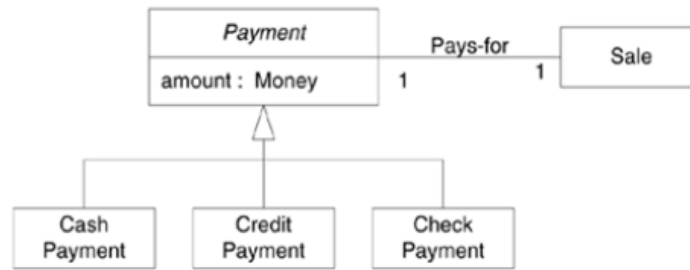


Fig: Subclass Conformance

- All Payments have an amount and are associated with sale.
- All Payment subclasses must conform to having an amount and paying for a sale- 100 % rule.
- 100 % of the conceptual superclass's definition should be applicable to the subclass. The subclass must conform to 100% of the superclass
 - ✓ Attribute
 - ✓ Association

CONCEPTUAL SUBCLASS SET CONFORMANCE

- A Conceptual subclass should be a member of the set of the superclass
- Conceptual subclass is a kind of superclass
- CreditPayment is a kind of Payment- is-a rule

CORRECT CONCEPTUAL SUBCLASS

A potential subclass should conform to the

- 1) 100% rule (Definition Conformance)
- 2) Is-a- Rule (Set membership conformance)

When to Define a Conceptual Subclass?

A **conceptual class** partition is a division of a conceptual class into disjoint subclasses.

Eg.

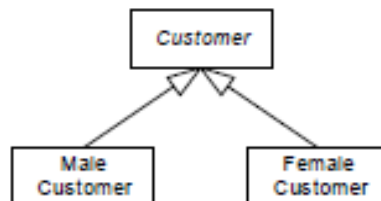


Fig: Conceptual class partition

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.

When to Define a Conceptual Superclass?

When commonality is identified among subclasses, generalization is done.

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.

Eg:

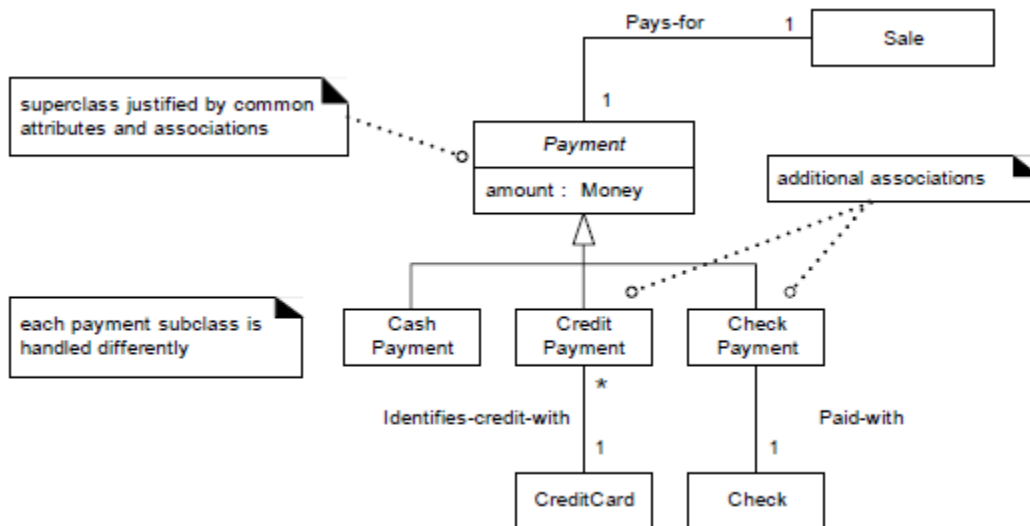


Fig: Justifying Payment Subclasses

ABSTRACT CONCEPTUAL CLASSES

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

Eg:

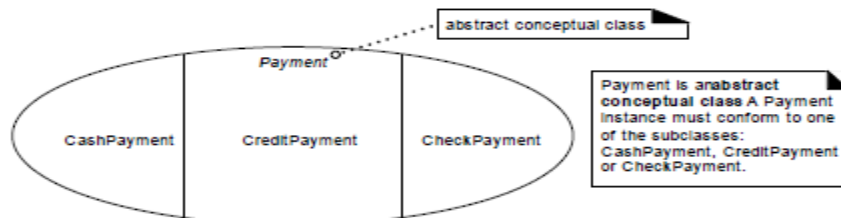
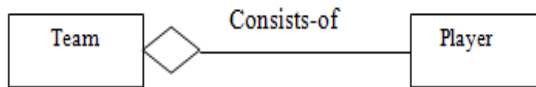


Fig: Abstract Conceptual Classes

AGGREGATION

Aggregation is a vague kind of association in the UML that loosely suggests whole-part relationships

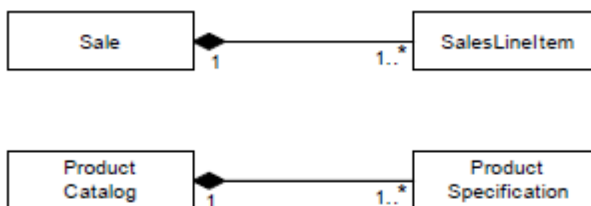
Eg:



COMPOSITION

Composition is a strong kind of whole-part aggregation and is useful to show in some other models.

Eg:



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.

SYSTEM SEQUENCE DIAGRAM (SSD)

- A **system sequence diagram** (SSD) is a picture that shows, for a particular scenario of a use case, the events that external actors generate, their order, and inter-system events.
- Sequence diagrams are an easy way of describing the behaviour of a system by viewing the interaction between the system and its environment.

Eg:

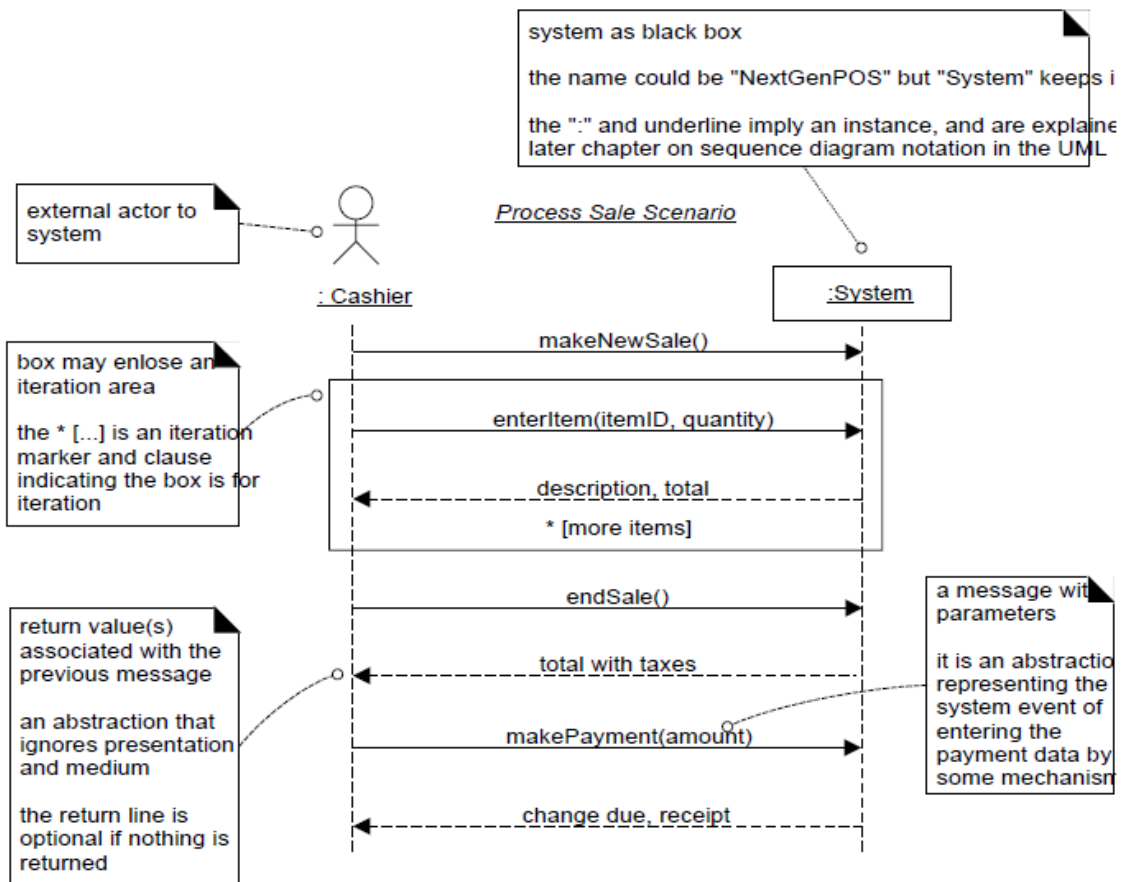


Fig: SSD for a Process Sale scenario

UML Perspective of Sequence Diagram

- Use Cases describe how external actors interact with the software system.
- During this interaction an actor generates system events to a system requesting for some system operation to handle the event.
- UML Sequence Diagrams are thereby a notation to depict actor interactions and the operations initiated by them.

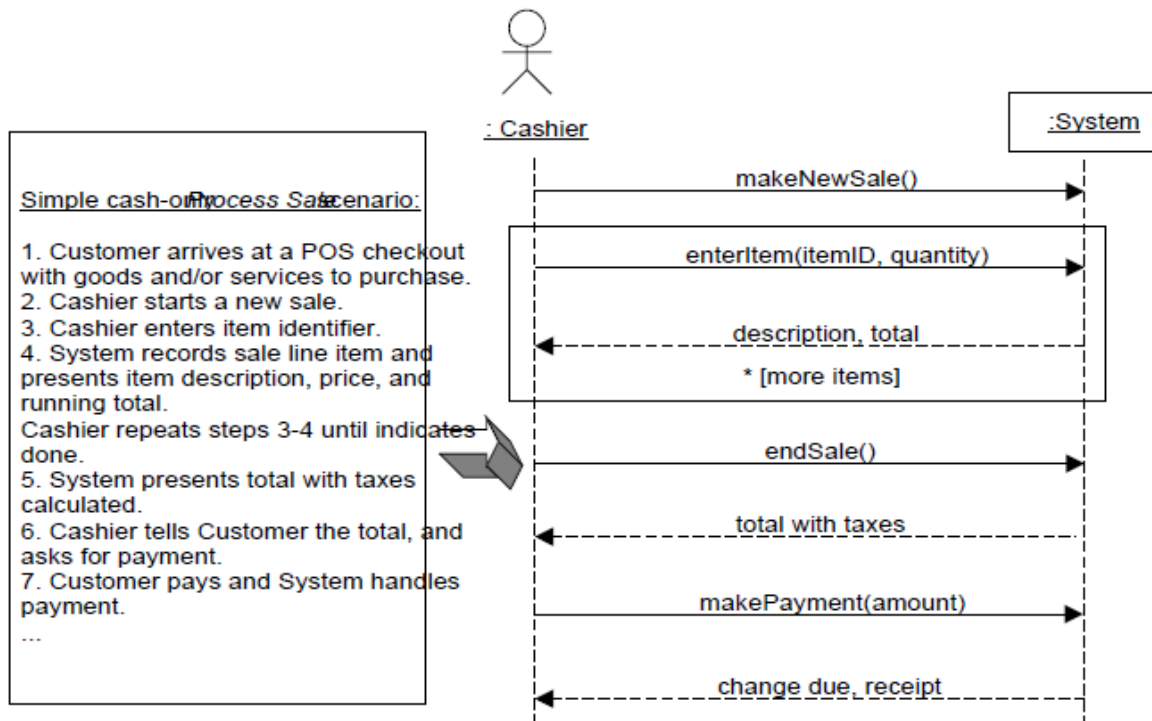
Why to draw SSD?

1. To investigate and define system behaviour before proceeding with a detailed design of how a software application will work.
System Behaviour-description of what a system does rather how it does. Such a kind of description is clearly depicted by system sequence diagram.
2. To know what events are coming into the system,so that software can be designed to handle those events and execute a response.

Three events that affects the software system

- i. External events from actors
- ii. Timer events
- iii. Faults or Exceptions

RELATIONSHIP BETWEEN SEQUENCE DIAGRAMS AND USE CASES



SYSTEM EVENTS AND SYSTEM BOUNDARY

System Event

A system event is an external event that directly stimulates the software.

System Boundary

The system boundary is usually chosen to be the software (and possibly hardware) system itself.

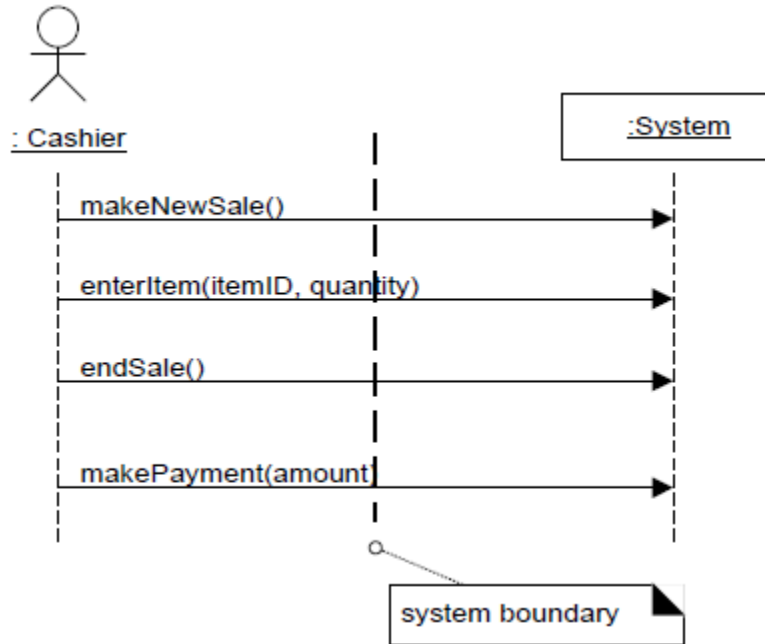


Fig: Defining System Boundary

NAMING SYSTEM EVENTS AND OPERATIONS

- System Events should be expressed at the abstract level of intention rather than in terms of the physical input device
- Choose event and operation names at an abstract level.

Eg:

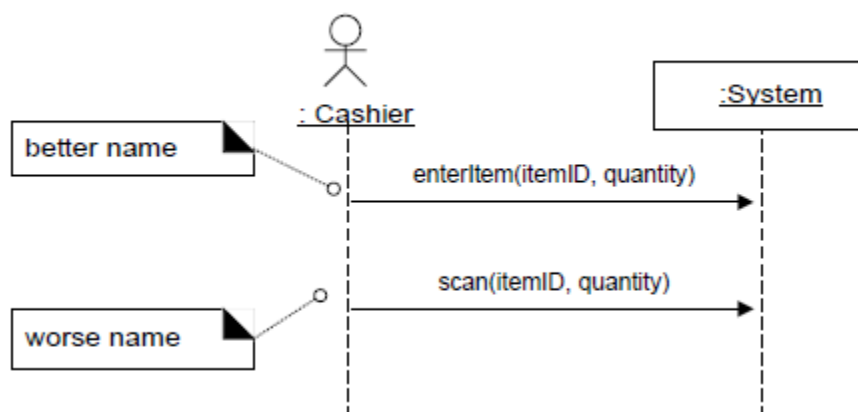


Fig: Naming System Events

Start the name of a system event with a verb such as add, enter, insert, make etc.

LOGICAL ARCHITECTURE AND UML PACKAGE DIAGRAMS

Logical Architecture

- Logical Architecture is the organization of software classes into packages, subsystems and layers.
- Logical Architecture does not depict how these elements are deployed across different operating system processes or across physical computers in a network.

UML Package Diagram

- UML Package diagrams are used to represent the logical architecture of a system- the layers, subsystems, packages.
- UML Package diagram is a way of grouping elements such as classes, other packages, use cases etc.

LOGICAL ARCHITECTURE REFINEMENT

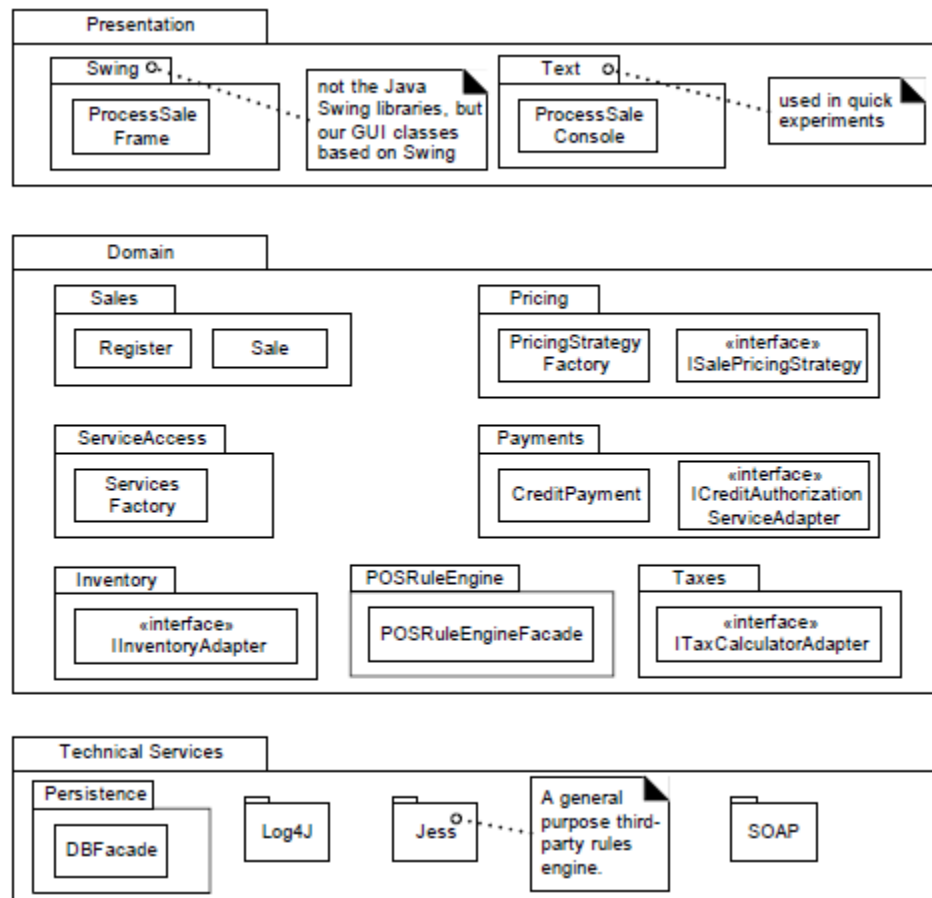


Fig: Partial Logical Layered Architecture in NextGen application

1. The Logical Architecture consists of three layers,
 - UI (User Interface) Layer
 - Domain Layer

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

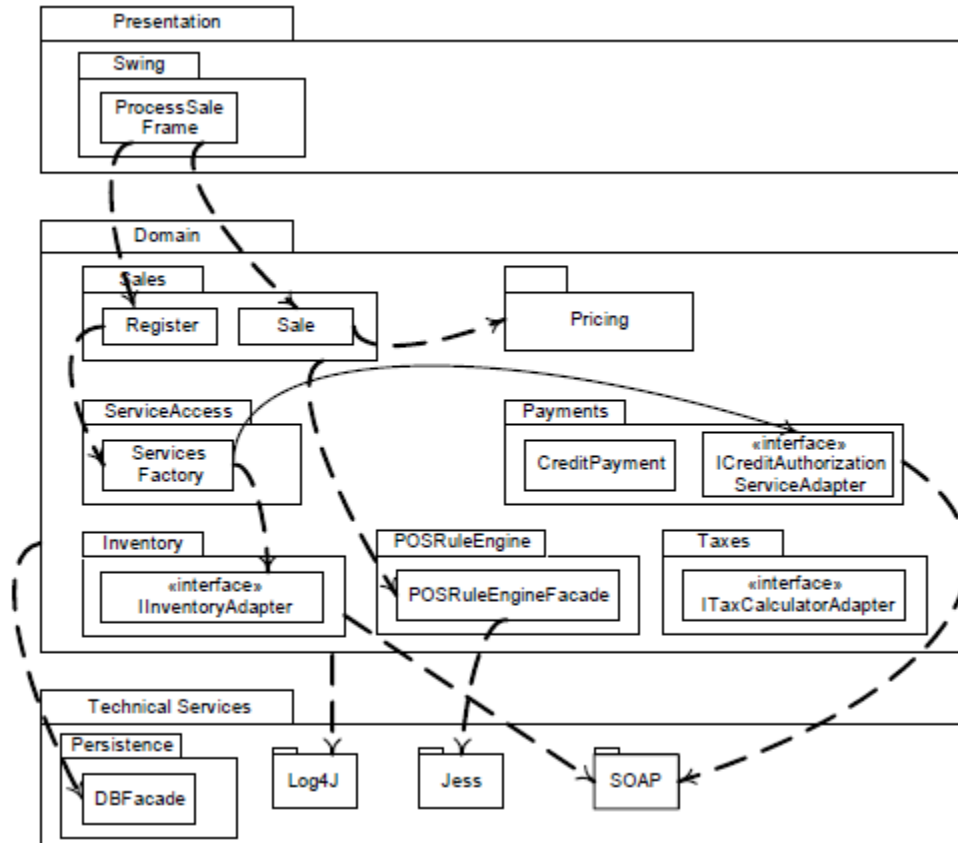


Fig: Coupling between layers and packages

1. Inter Layer and Inter Package Coupling

1. Dependency lines are used to communicate coupling between packages or types in packages.
2. Many elements of the packages 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 PosRule Engine to Jess
 - e. From Inventory to SOAP etc.

2. Partial Package Coupling

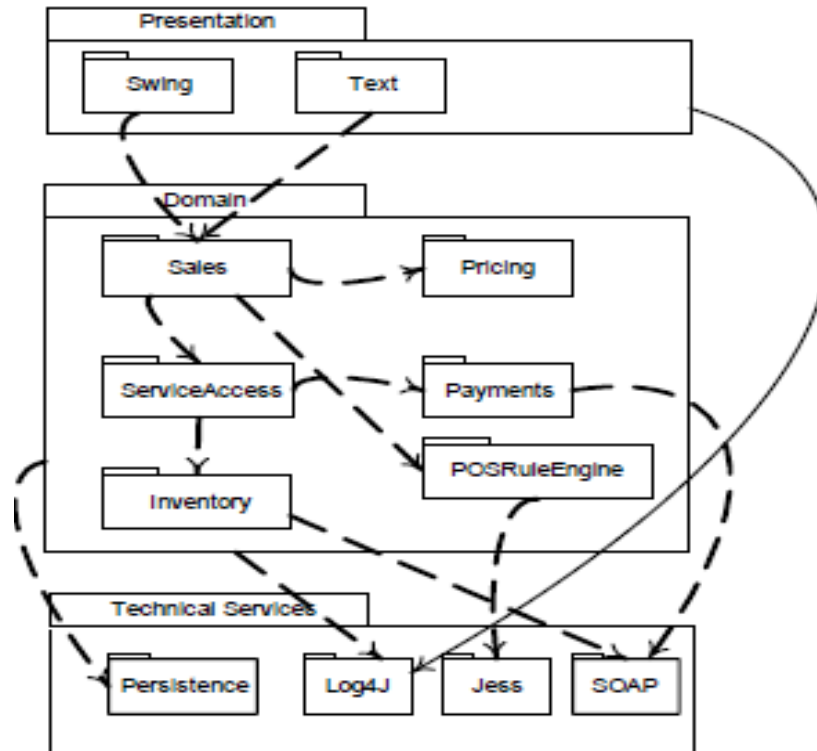


Fig: Partial Package Coupling

3. Inter Layer and Inter Package Interaction Scenarios

- Package diagrams are static in nature
- To understand the dynamic actions i.e how the objects across the layers connect and communicate with each other, interaction diagrams should be drawn
- Logical View of the architecture focuses on the collaborations as they cross the layer and package boundaries
- A set of interaction diagrams illustrate architecturally significant scenarios that depicts the large scale or big ideas in design

Applying UML

- The package of a type can optionally be shown by qualifying the type with UML path name expression <package Name> :: <Type Name>
Eg: Domain :: Sales :: Register
- Subsystem stereotype (<<subsystem>>) used in the diagram is a special kind of package with behaviour and interfaces.

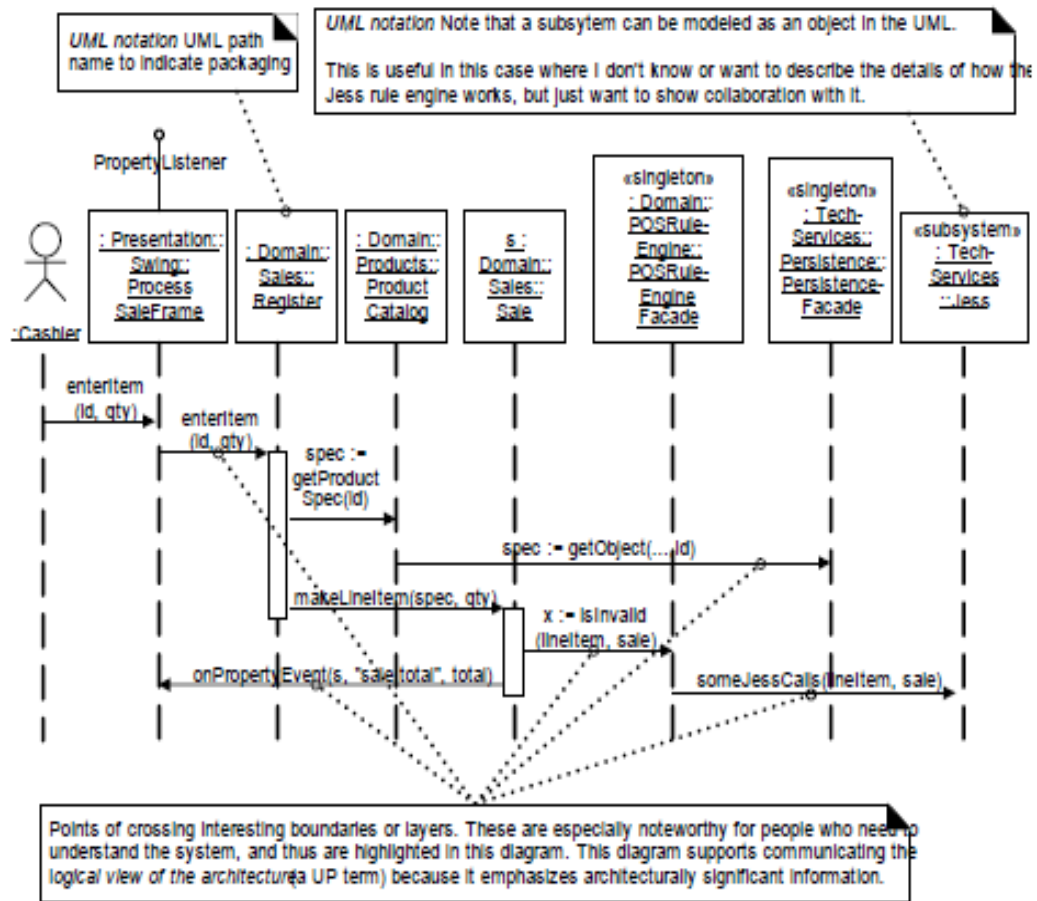


Fig: An architecturally significant interaction diagram

COLLABORATIONS WITH LAYERS PATTERN

Two design decisions at an architectural level

1. What are big parts?
2. How they are connected?

Simple packages versus Subsystems

1. **Packages** groups the factory and strategies used
 Packages are used to represent the different layers of source code.
Eg: Pricing, Payroll, Foundation Packages such as java.util
2. **Subsystems** are a type of stereotyped component that represent independent, behavioural units in a system.
Eg: Persistence, POS Rule Engine, Jess

Façade

- Façade is the most common pattern of access for packages that represent subsystems.
- Façade is a GoF design pattern

- Public façade object defines the services for the subsystem and clients collaborate with the façade not internal subsystem components.

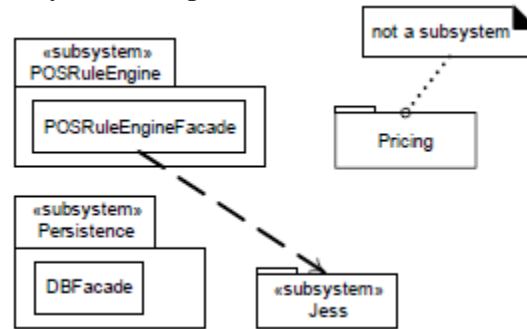
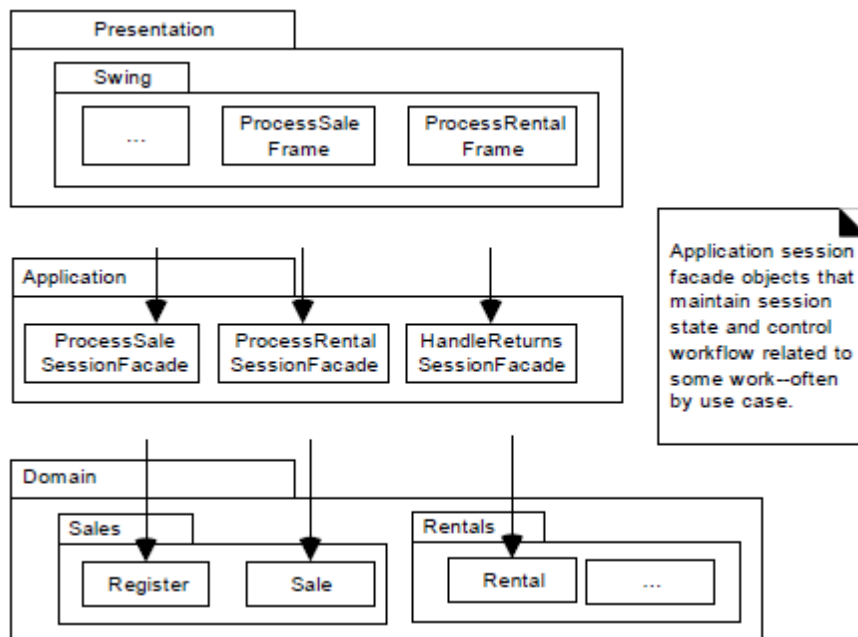


Fig: Subsystem Stereotype

Session Facades and the application layer



Session façade is the one where each session instance represents a session with one client.

Controller

- Controller Pattern describes common choices in client side handlers for system operation requests emitting form the UI layer.

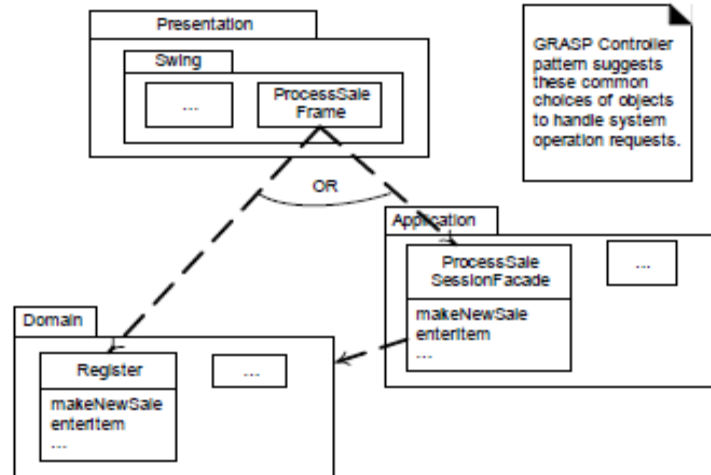


Fig: Controller

System Operations and Layers

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.

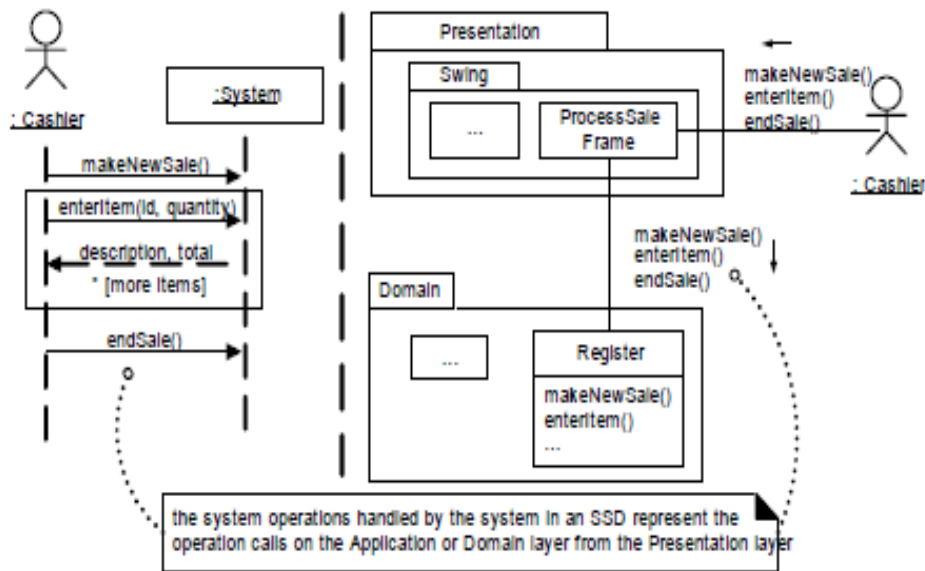


Fig: System Operations in SSD using Layers

Upward Collaboration with observer

Observer Pattern is used for upward collaboration i.e when the lower application or domain layer needs to communicate with the upward UI Layer via observer pattern.

1. UI objects in the higher Presentation layer implement an interface such as Property Listener or Alarm Listener, and are subscribers or listeners to events coming from objects in the lower layers.

- The lower layer objects directly sends messages to the upper layer UI objects. Coupling takes place only to the objects viewed as things that implement an interface.

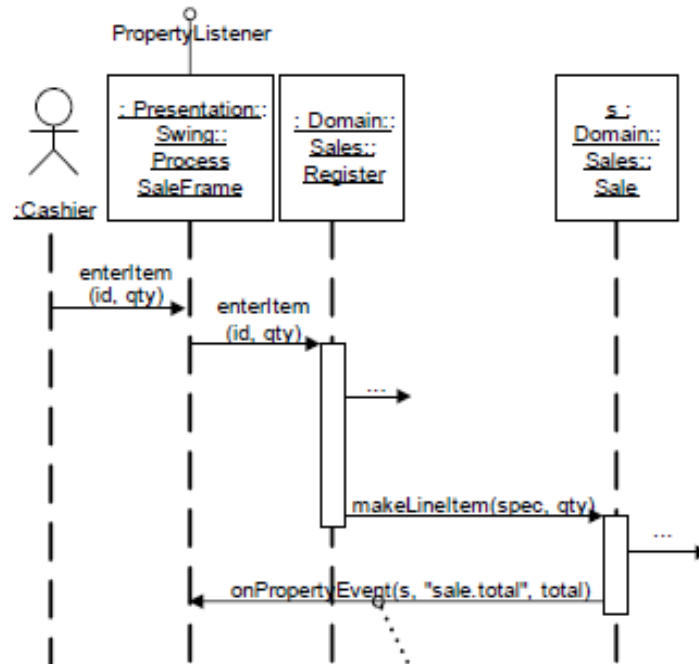


Fig: Observer Pattern

UML CLASS DIAGRAM:

Class diagram is a static diagram. It represents the static view of an application. The class diagram describes the attributes and operations of a class and also the constraints imposed on the system. The class diagrams are widely used in the modeling of object oriented systems because they are the only UML diagrams which can be mapped directly with object oriented languages.

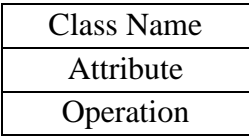
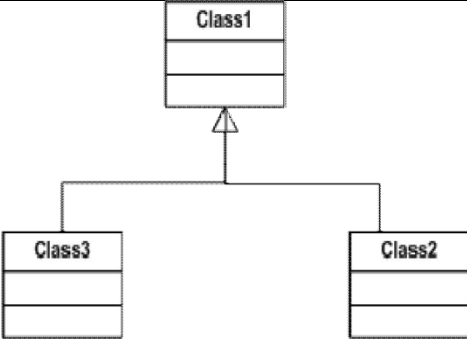
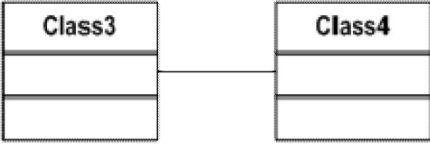
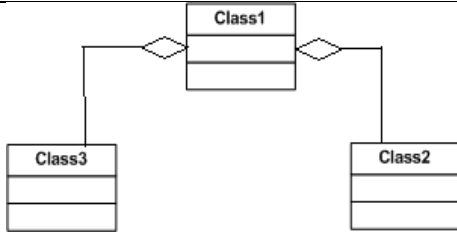

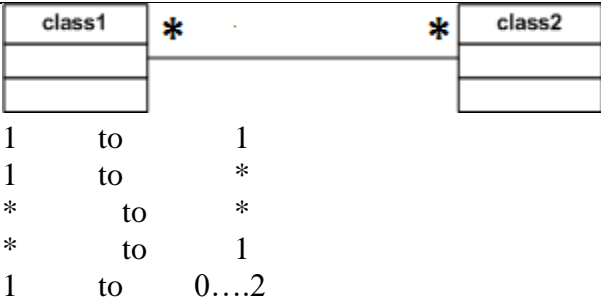
Purpose:

- Analysis and design of the static view of an application
- Describe responsibilities of a system
- Base for Component and Deployment Diagrams
- Forward and Reverse Engineering

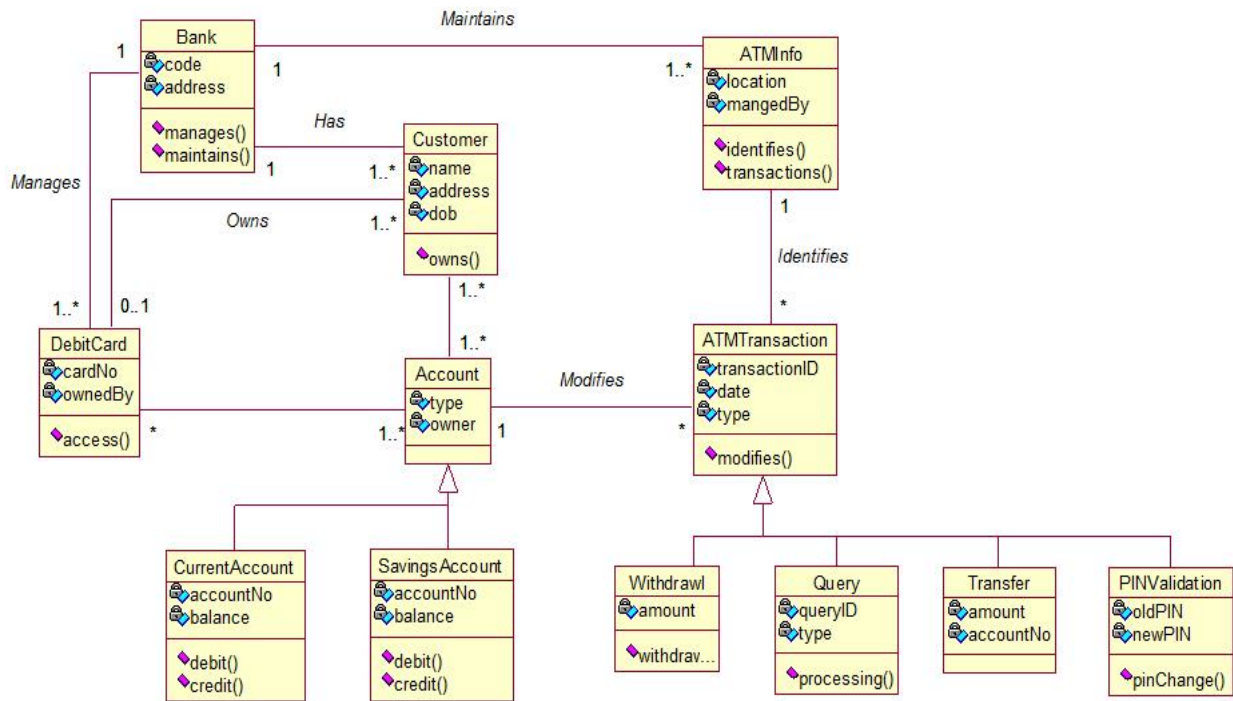
Uses:

- Describes the static view of the system
- Shows the collaboration among the elements of the static view
- Describes the functionalities performed by the system.
- Construction of software applications using object oriented languages.

Notations:

S.No	Name	Notation	Description
1	Class		Class is an entity which describes a group of objects with same properties & behavior.
2	Generalization		Generalization refers to a relationship between two classes where one class is a specialized version of another.
3	Association		Association represent static relationships between classes.
4	Aggregation		Aggregation is a vague kind of association in the UML that loosely suggests whole-part relationships.
5	Composition		Composition is a strong kind of whole-part aggregation.
6	Multiplicity	 <p>1 to 1 1 to * * to * * to 1 1 to 0...2</p>	Multiplicity specifies the number of instances of one class that may relate to a single instance of an associated class.

Sample Example – ATM System



UML INTERACTION DIAGRAM

Interaction diagrams are used to visualize the interactive behavior of the system. The Interactive behaviour is represented in UML by two diagrams namely,

- **Sequence Diagram-** It emphasizes on time sequence of messages
- **Collaboration Diagram-** It emphasizes on structural organization of the objects that send and receive messages.

Purpose:

5. To capture dynamic behaviour of a system
6. To describe the message flow in the system
7. To describe structural organization of the objects
8. To describe interaction among objects

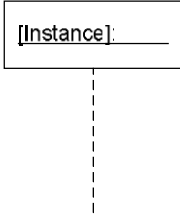
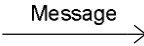
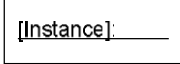
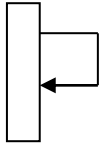
III. SEQUENCE DIAGRAM

Sequence diagram describes an interaction by focusing on the sequence of messages that are exchanged, along with their corresponding occurrence specifications on the lifelines.

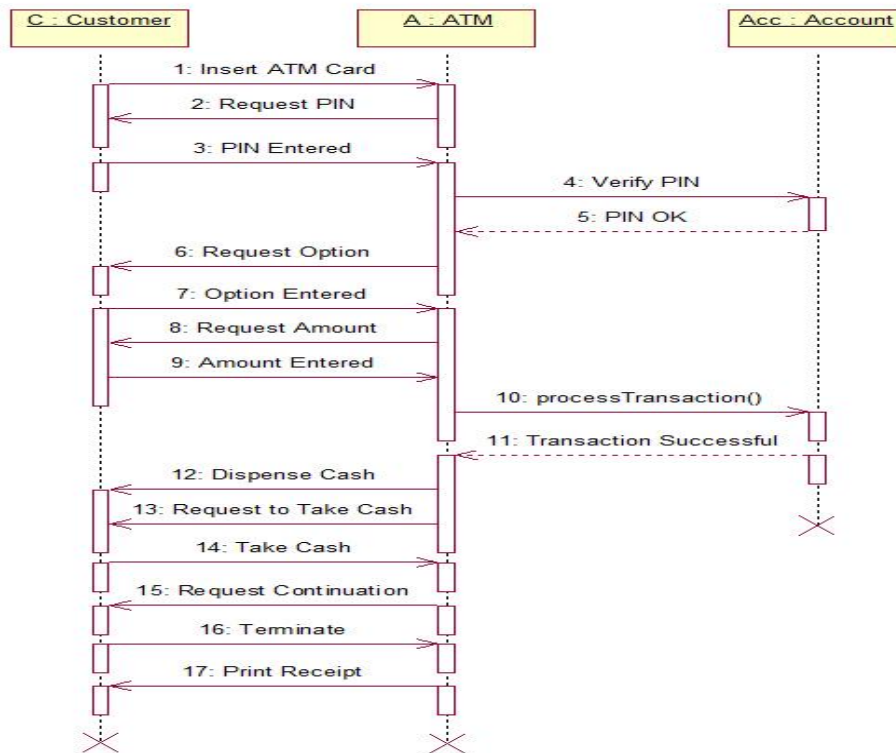
Uses:

5. To model flow of control by time sequence
6. To model flow of control by structural organizations
7. Forward engineering
8. Reverse engineering

Notations:

S.No	Name	Notation	Description
1	Lifeline		Lifeline represents the duration during which an object is alive and interacting with other objects in the system.
2	Message		To send message from one object to another.
3	Object		It represents the existence of an object of a particular time.
4	Self message		Self message is a message by the object to itself.

Sample Example – ATM System



IV. COLLABORATION DIAGRAM

Collaboration or Communication diagram is also used to model the dynamic behaviour of the system. It emphasizes on structural organization of the objects that send and receive messages.

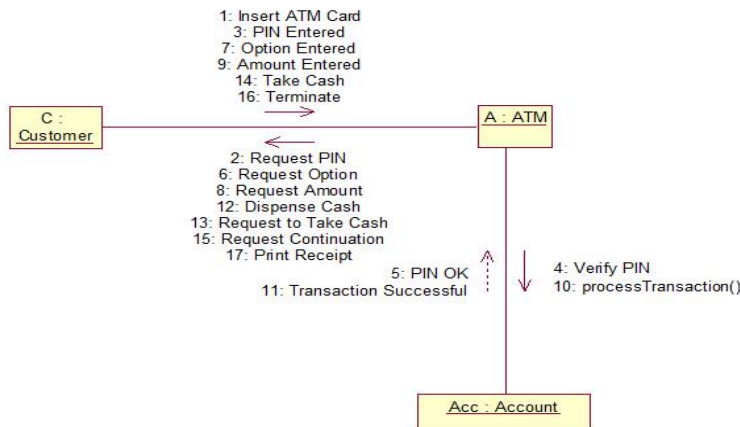
Uses:

4. Used to show the messages that flow from one object to another within the system and the order in which they happen.
5. Used to track the source of the message from where it has been sent
6. Used to provide relationships and interactions among software objects

Notations:

S.No	Name	Notation	Description
1	Link	_____	A Link is a connection path between two objects
2	Message		Communication between objects takes place through messages. A sequence number is added to show the sequential order of messages.
3	Message Number Sequencing		Numbers included along with the messages indicate the order of the message in an interaction.

Sample Example – ATM System



APPLYING GoF DESIGN PATTERNS

1. ADAPTER (GoF)

Adapter pattern works as a bridge between two incompatible interfaces. It is used to convert the programming interface of one class into that of another.

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.

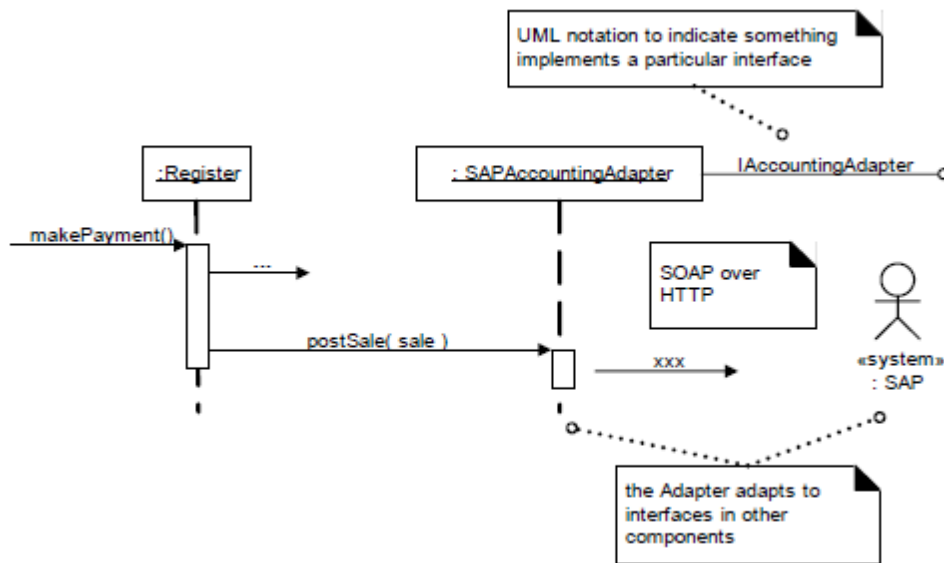


Fig: Adapter Pattern

2. SINGLETON (GoF)

- Singleton Pattern is a design pattern used to implement the mathematical concept of a singleton, by restricting the instantiation of a class to one object.
- Singleton is a class which only allows a single instance of itself to be created and usually give simple access to that 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

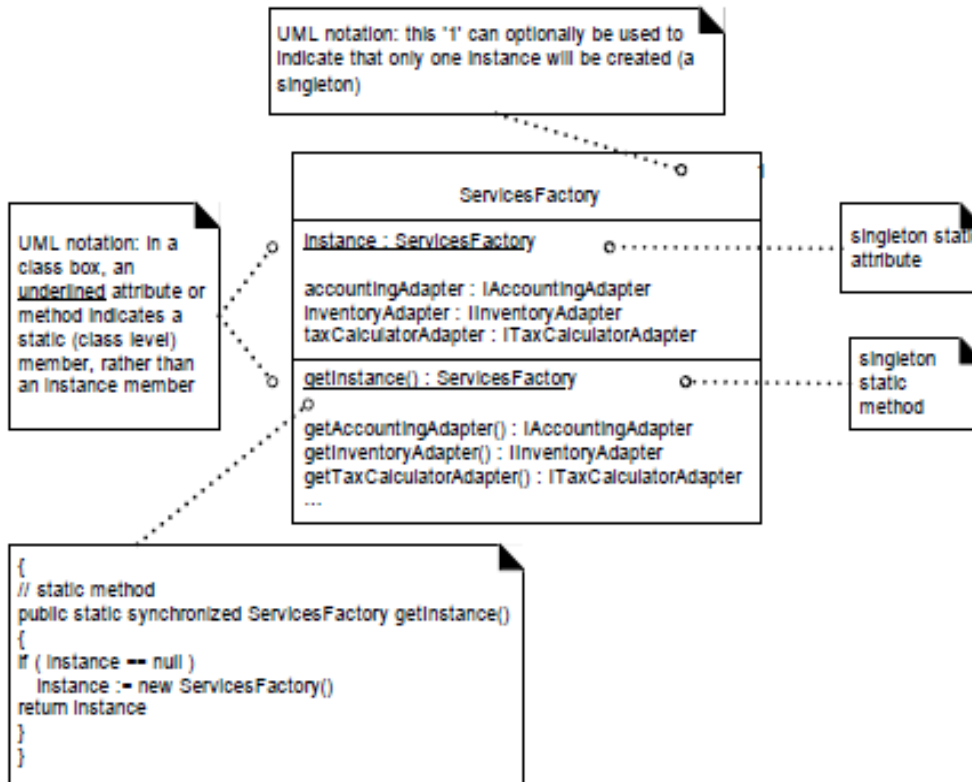


Fig: Singleton Pattern

3. FACTORY PATTERN

Name: Factory

Problem: Who should be responsible for creating objects when there are special considerations, such as complex creation logic, a desire to separate the creation responsibilities for better cohesion, and so forth?

Solution: Create a Pure Fabrication object called a Factory that handles the creation.

Advantages of Factory objects

- ✓ Separate the responsibility of complex creation into cohesive helper objects.
- ✓ Hide potentially complex creation logic.
- ✓ Allow introduction of performance-enhancing memory management strategies, such as object caching or recycling.

In the below diagram, In ServicesFactory, the logic to decide which class to create is resolved by reading in class name from an external source and then dynamically loading the class. This is termed as **Partial Data Driven Design**.

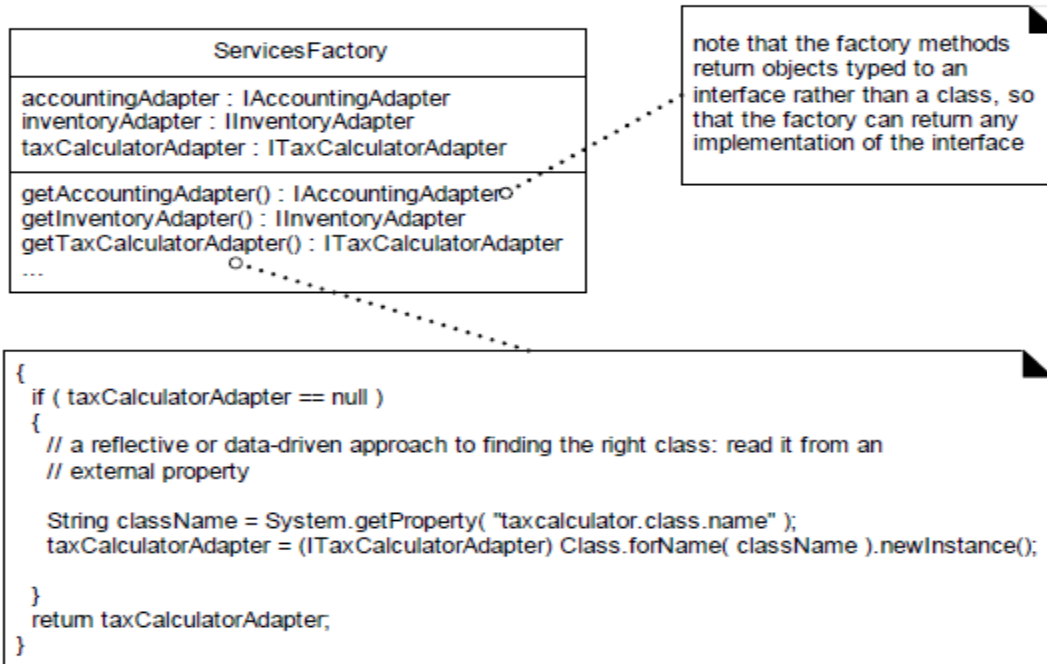


Fig: Factory Pattern

4. OBSERVER PATTERN

Problem

Different kinds of subscriber objects are interested in state changes or events of a publisher object, and want to react in their own unique way when the publisher generates an event. Moreover, the publisher wants to maintain low coupling to the subscribers. What to do?

Solution:

Define a “subscriber” or “listener” interface. Subscribers implement this interface. The publisher can dynamically register subscribers who are interested in an event and notify them when an event occurs.

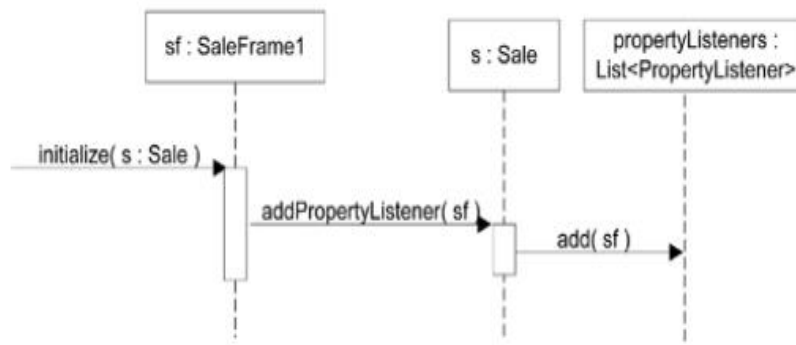


Fig: The observer `SaleFrame1` subscribes to the publisher `Sale`

The SaleFrame1 object is the observer/subscriber/listener. In the above diagram it subscribes to interest in property events of the Sale, which is a publisher of property events. The Sale adds the object to its list of PropertyListener subscribers. The Sale does not know about the SaleFrame1 as a SaleFrame1 object, but only as a PropertyListener object, this lowers the coupling from the model up to the view layer.

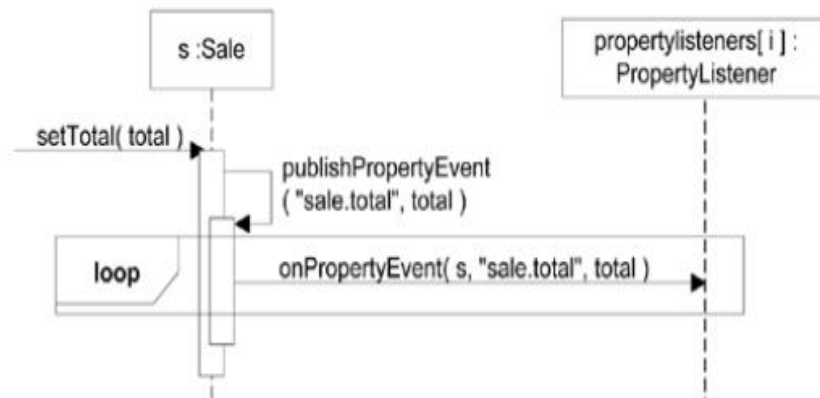


Fig: The Sale publishes a property event to all its subscribers

In the above diagram, when the sale total changes, it iterates across all its registered subscribers, and “publishes an event” by sending the onPropertyEvent message to each.

UNIT –V

CODING AND TESTING

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

MAPPING DESIGN TO CODE

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

- Class and Interface Definitions
- Method Definitions

Creating Class Definitions from DCD’s (Design Class Diagrams)

DCD’s depict the class or interface name, superclasses, method signatures, and simple attributes of a class.

Defining a Class with Method Signature and Attributes

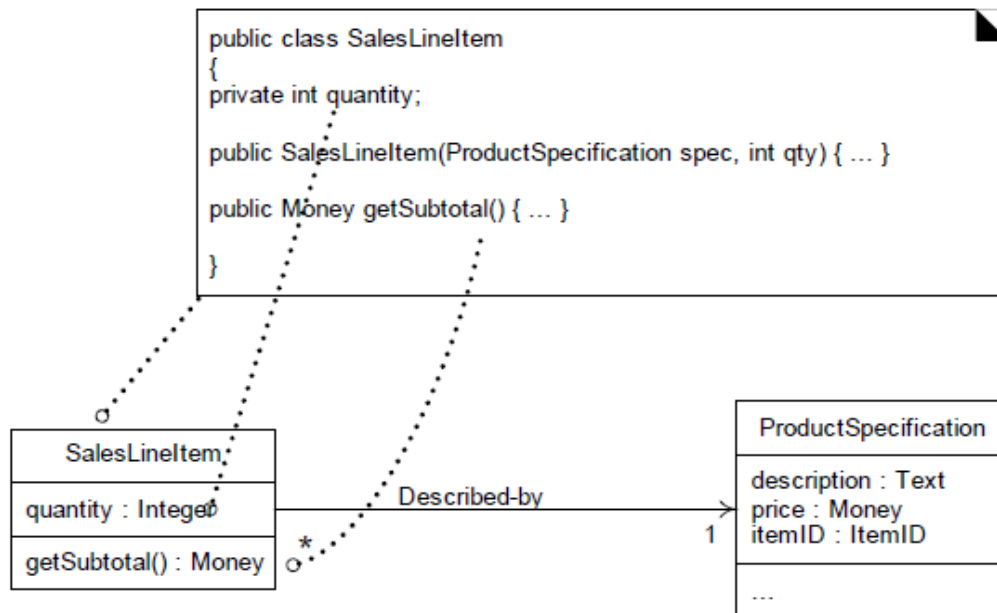


Fig: SalesLineItem in Java

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.

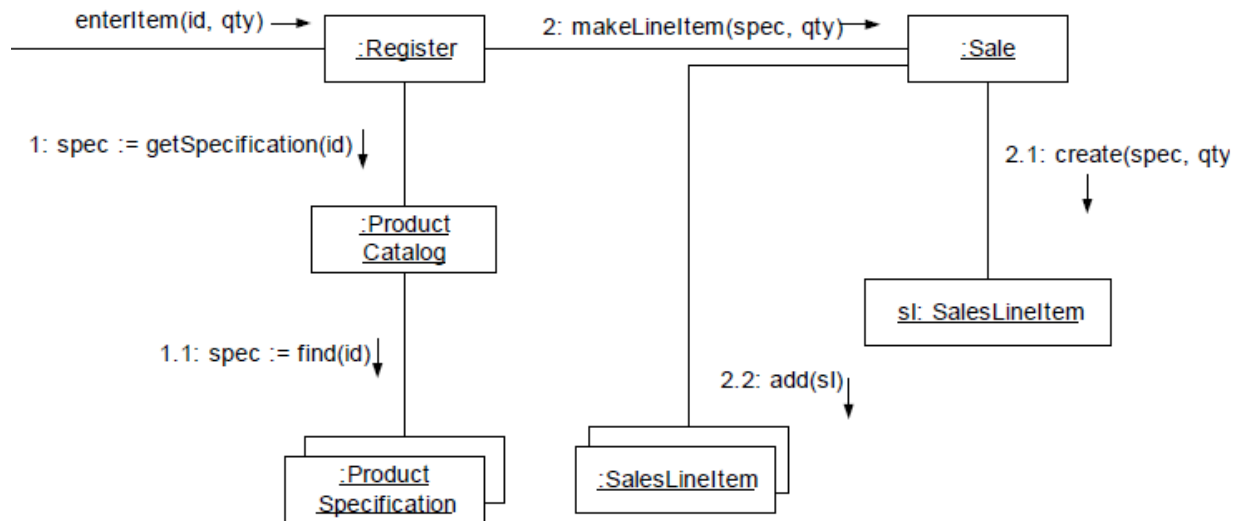


Fig: Interaction Diagram depicting enterItem message sent to Register instance

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*.

```
ProductSpecification spec = catalog. getSpecification( itemID );
```

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

```
sale .makeLineItemf spec, qty);
```

Each sequenced message within a method, as shown on the interaction diagram, is mapped to a statement in the Java method.

The Register-enterItem Method

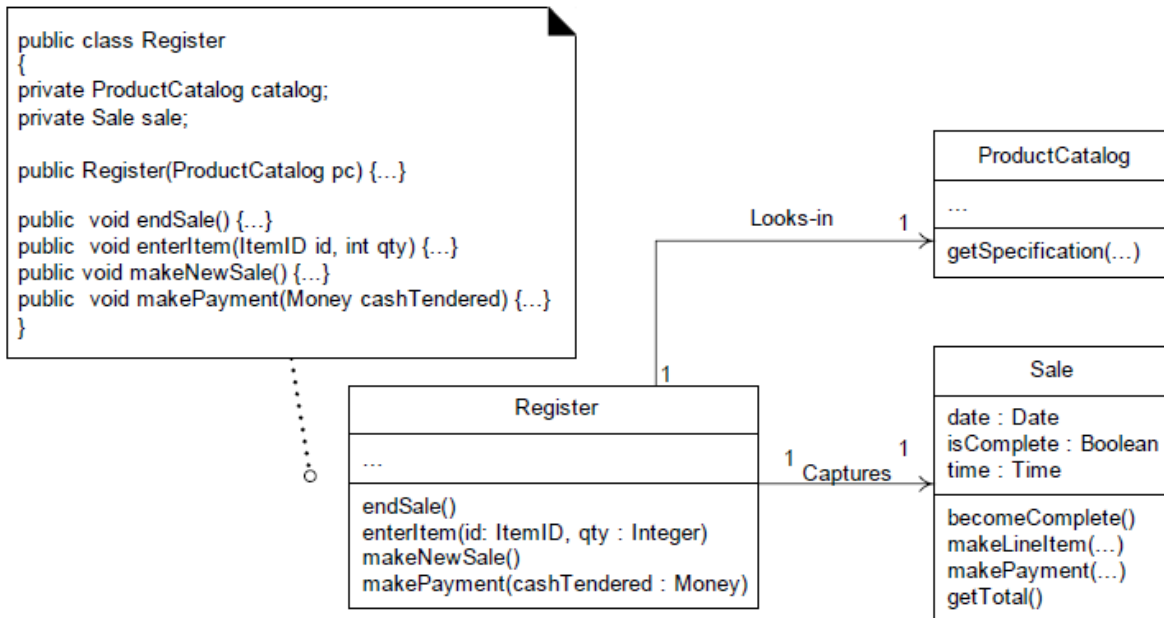


Fig: Register Class

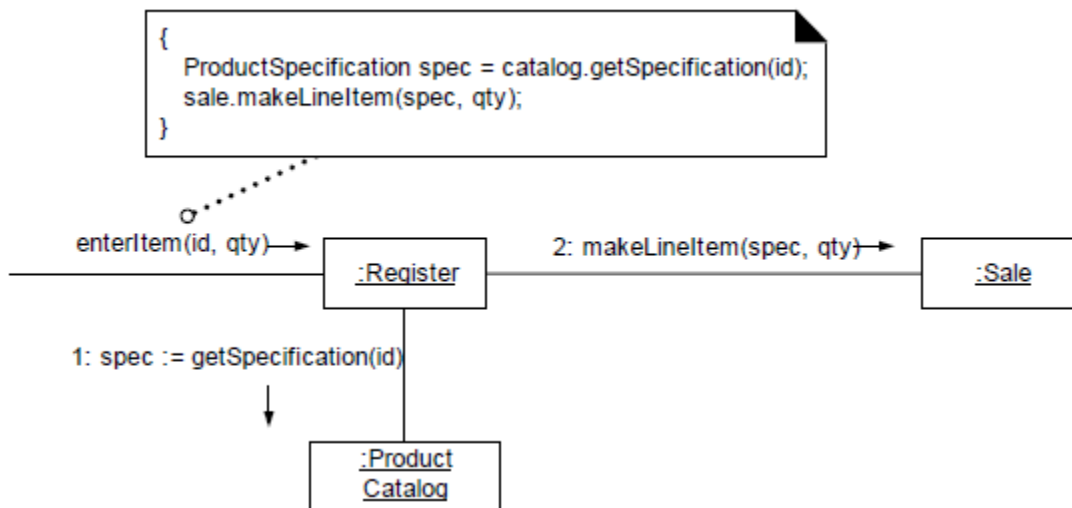


Fig: The enterItem method

Collection Classes in Code

A collection class is a container which holds a number of items in a data structure and provides various operations to manipulate the contents of the collection.

Collection object examples- List, Map etc.

Java libraries contain collection classes such as ArrayList and Hash Map which implement the List and Map interfaces.

Eg: Using ArrayList, the sale class can define an attribute that maintain an ordered List of SalesLineItem instances.

Exceptions and Error Handling

Exception handling is a programming language construct to handle the occurrence of exceptions, special conditions that change the normal flow of program execution. The point of exception handling routines is to ensure that the code can handle error conditions. In terms of UML, exceptions can be indicated in the property strings of messages and operation declarations.

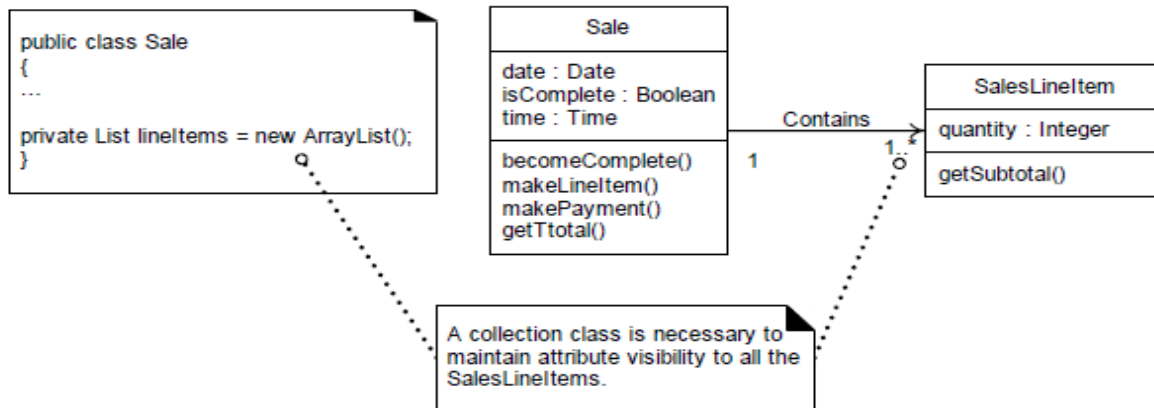


Fig: Adding a collection

Defining the Sale.makeLineItem Method

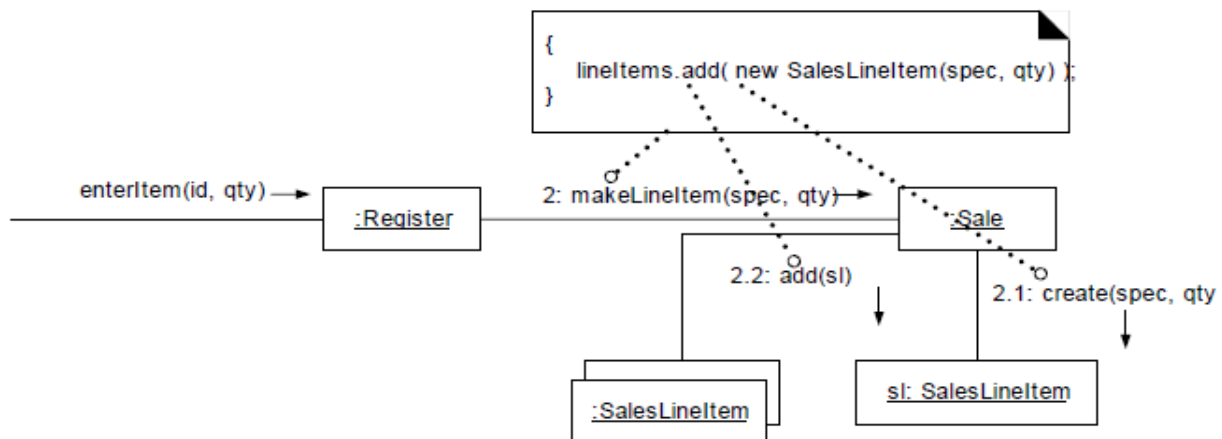


Fig: Sale.makeLineItem method

Order of Implementation

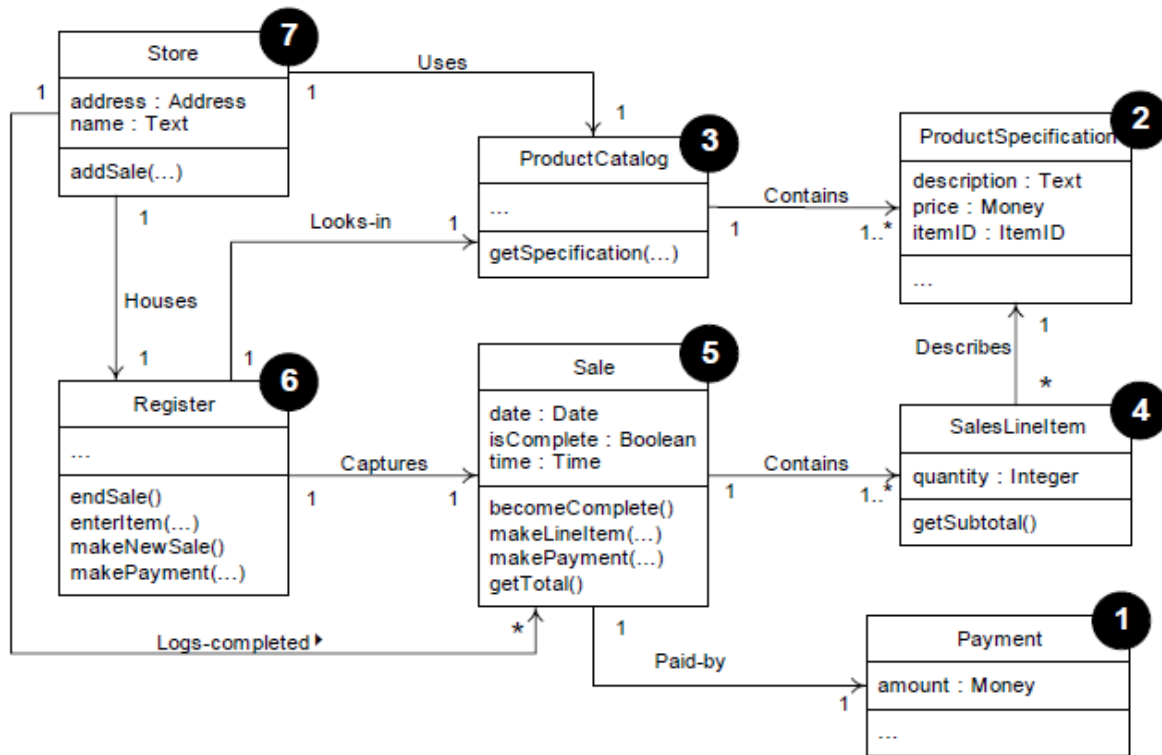


Fig: Possible order of class implementation and testing

Test Driven or Test First Development (TDD)

- **TDD** is a software development process where the developers first writes a failing automated test case that defines a desired improvement or new function, then produces code to pass that test and finally refactors the new code to acceptable standards.
- TDD requires developers to create automated unit tests that define code requirements before writing the code itself.
- In OO unit testing TDD style, test code is written before the class to be tested and the developer writes unit testing code for nearly all production code.

Refactoring

Refactoring is a structured, disciplined method to rewrite or restructure existing code without changing its external behaviour, applying small transformation steps combined with re-executing tests each step.

NextGen POS Program Solution

Class Payment

```
public class Payment {
    private Money amount;
    public Payment( Money cashTendered ){ amount = cashTendered; }
    public Money getAmount() { return amount; } }
```

Class ProductCatalog

```
public class ProductCatalog {
    private Map productSpecifications = new HashMap();

    public ProductCatalog() {
        // sample data
        ItemID id1 = new ItemID( 100 );
        ItemID id2 = new ItemID( 200 );
        Money price = new Money( 3 );

        ProductSpecification ps;
        ps = new ProductSpecification( id1, price, "product 1" );
        productSpecifications.put( id1, ps );
        ps = new ProductSpecification( id2, price, "product 2" );
        ProductSpecifications.put( id2, ps ); }

    public ProductSpecification getSpecification( ItemID id ) {
        return (ProductSpecification)productSpecifications.get( id );
    }
}
```

TESTING

- Testing is an activity to check whether the actual results match the expected results and to ensure that the system is defect free.
- Testing also helps to identify errors, gaps or missing requirements in contrary to the actual requirements. It can be either done manually or using automated tools.

ISSUES IN OO TESTING

Traditional testing methods are not directly applicable to OO programs as they involve OO concepts including encapsulation, inheritance, and polymorphism. These concepts lead to issues, which are yet to be resolved. Some of these issues are listed below.

1. Basic unit of unit testing

- The class is natural unit for unit test case design.
- The methods are meaningless apart from their class.
- Testing a class instance (an object) can validate a class in isolation.

- When individually validated classes are used to create more complex classes in an application system, the entire subsystem must be tested as whole before it can be considered to be validated (integration testing).

2. Implication of Encapsulation

- Encapsulation of attributes and methods in class may create obstacles while testing. As methods are invoked through the object of corresponding class, testing cannot be accomplished without object.
- In addition, the state of object at the time of invocation of method affects its behavior. Hence, testing depends not only on the object but on the state of object also, which is very difficult to acquire.

3. Implication of Inheritance.

- Inheritance introduce problems that are not found in traditional software.
- Test cases designed for base class are not applicable to derived class always (especially, when derived class is used in different context). Thus, most testing methods require some kind of adaptation in order to function properly in an OO environment.

4. Implication of Genericity

- Genericity is basically change in underlying structure.
- We need to apply white box testing techniques that exercise this change.

5. Implications of Polymorphism

- Each possible binding of polymorphic component requires a separate set of test cases.
- Many server classes may need to be integrated before a client class can be tested.
- It is difficult to determine such bindings.
- It complicates the integration planning and testing.

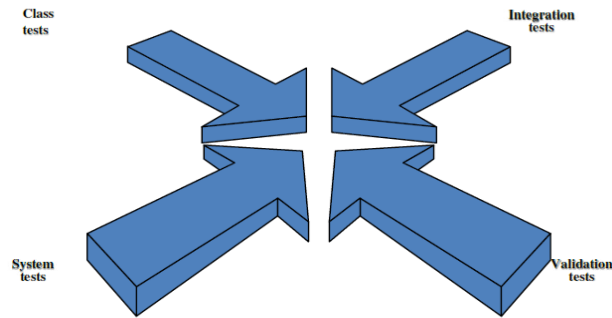
6. Implications for testing processes

- Re-examine all testing techniques and processes.

CLASS TESTING

- 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.

Testing OO Code



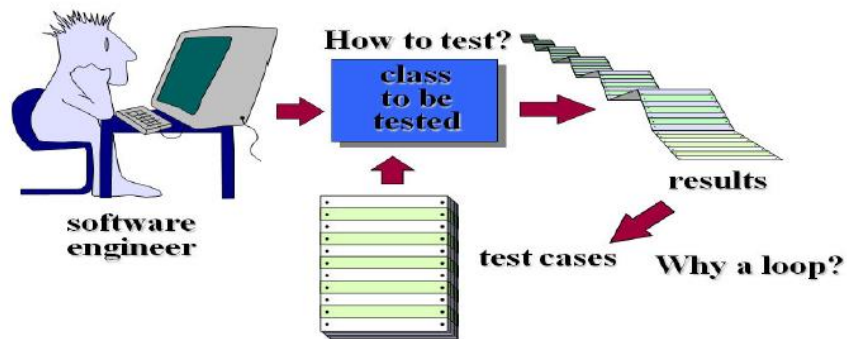
Approach:

- 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. A list of messages and operations that will be exercised as a consequence of the test.
 - b. A list of exceptions that may occur as the object is tested.
 - c. 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)
 - d. 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 class state.
- **Inheritance and polymorphism:**
 1. Each new context of use (subclass) requires re-testing because a method may be implemented differently (polymorphism).
 2. Other unaltered methods within the subclass may use the redefined method and need to be tested.
- **White box tests:**
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.
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 possible defects that may be encountered in operation calls and message connections and builds appropriate test cases
 - misses incorrect specification and errors in subsystem interactions.

OO INTEGRATION TESTING

Integration testing is the phase in software testing in which individual software modules are combined and tested as a group. Integration testing takes as its input modules that have been unit tested, groups them in larger aggregates, applies tests defined in an integration test plan to those aggregates, and delivers as its output the integrated system ready for system testing.

Kinds of integration testing:

- **big bang testing** - An inappropriate approach to integration testing in which you take the entire integrated system and test it as a unit

- **incremental testing** - 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 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.

GUI TESTING

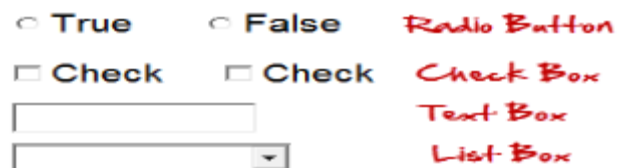
GUI testing is the process of ensuring proper functionality of the graphical user interface (GUI) for a given application and making sure it conforms to its written specifications. In addition to functionality, GUI testing evaluates design elements such as layout, colors, fonts, font sizes, labels, text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

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 to understand the UI. If a user is not comfortable with the Interface or find Application complex to understand he/she would never going to use that Application again.

What do you check in GUI Testing?

The following checklist will ensure detailed GUI Testing.

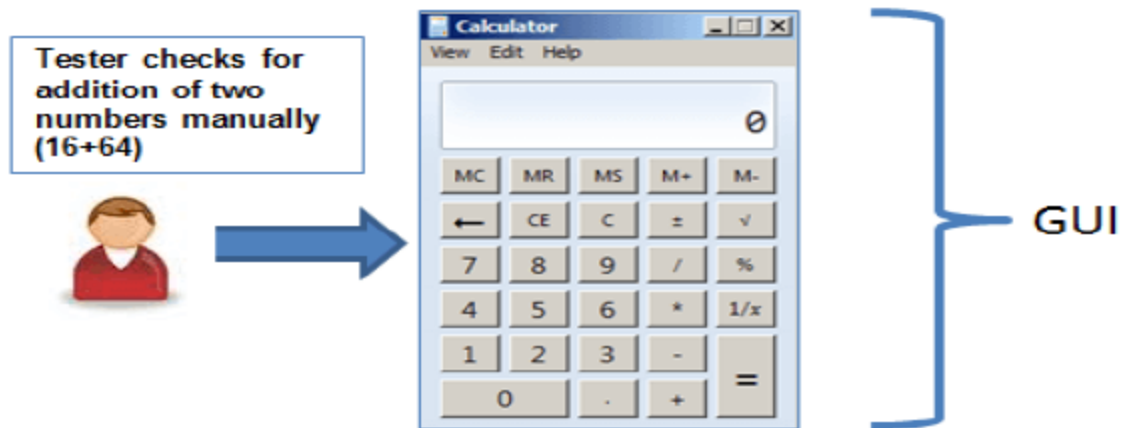
- Check all the GUI elements for size, position, width, length and acceptance of characters or numbers. For instance, you must be able to provide inputs to the input fields.
- Check if you can execute the intended functionality of the application using the GUI
- Check Error Messages are displayed correctly
- Check for Clear separation 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
- 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:

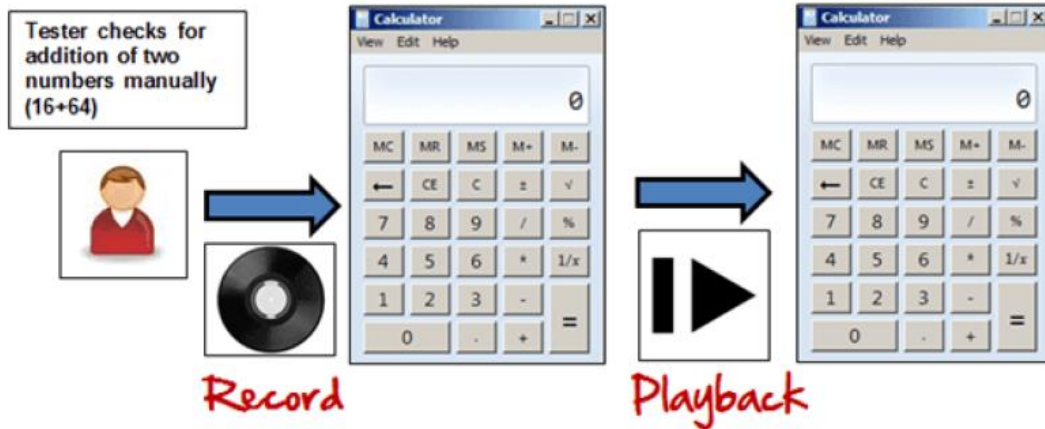
Manual Based Testing

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



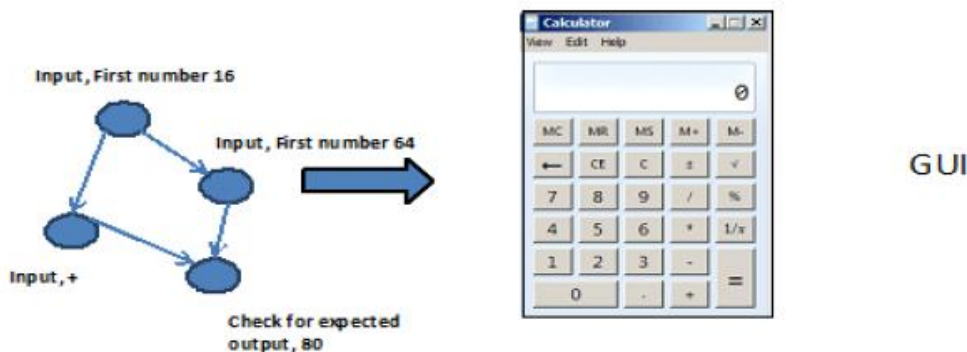
Record and Replay

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 is an application of model-based design for designing and optionally also executing artifacts to perform software testing or system testing. Models can be used to represent the desired behavior of a system under test (SUT), or to represent testing strategies and a test environment.



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.

OO SYSTEM TESTING

System testing of software or hardware is testing conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements. System testing falls within the scope of black-box testing, and as such, should require no knowledge of the inner design of the code or logic.

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.
- 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
- **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.