Department of Information Technology

Lab Manual

BE IT

Object Oriented Analysis & Modeling

MGM's

Jawaharlal Nehru Engineering College, Aurangabad

FOREWORD

It is my great pleasure to present this laboratory manual for Final Year engineering students for the subject of Object Oriented Analysis & Modeling keeping in view the vast coverage required for analysis and design of software systems.

As you may be aware that MGM has already been awarded with ISO 9000 certification and it is our aim to technically equip students taking the advantage of the procedural aspects of ISO 9000 Certification.

Faculty members are also advised that covering these aspects in initial stage itself, will relive them in future as much of the load will be taken care by the enthusiastic energies of the students once they are conceptually clear.

Dr. H.H.Shinde

Principal

LABORATORY MANUAL CONTENTS

This manual is intended for the Final Year students of Information Technology in the subject of Object Oriented Analysis & Modeling. This manual typically contains practical/Lab Sessions related to the subject to enhance understanding.

The time spent on getting the design right before you start programming will almost always save you time in the end. It's much, much easier to make major changes on a design, which is after all just squiggly lines on paper, and then it is to make changes in hundreds or thousands of lines of code.

The design process is typically split into distinct phases: Object-Oriented Analysis (OOA) and Object Oriented Design (OOD) and Design Pattern.

Students are advised to thoroughly go though this manual rather than only topics mentioned in the syllabus as practical aspects are the key to understanding and conceptual visualization of theoretical aspects covered in the books.

Good Luck for your Enjoyable Laboratory Sessions

Dr. S.C. Tamane — S. N. Bhasme

HOD, IT — Asst. Prof., IT Dept

DOs and DON'Ts in Laboratory:

1. Make entry in the Log Book as soon as you enter the Laboratory.

2. All the students should sit according to their roll numbers starting from their left to right.

3. All the students are supposed to enter the terminal number in the log book.

4. Do not change the terminal on which you are working.

5. All the students are expected to get at least the algorithm of the program/concept to be implemented.

6. Strictly observe the instructions given by the teacher/Lab Instructor.

Instruction for Laboratory Teachers::

1. Submission related to whatever lab work has been completed should be done during the next lab session. The immediate arrangements for printouts related to submission on the day of practical assignments.

2. Students should be taught for taking the printouts under the observation of lab teacher.

3. The promptness of submission should be encouraged by way of marking and evaluation patterns that will benefit the sincere students.

MGM's

Jawaharlal Nehru Engineering College, Aurangabad

Department of Information Technology

Vision of IT Department:

To develop computer engineers with necessary analytical ability and human values who can creatively design, implement a wide spectrum of computer systems for welfare of the society.

Mission of the IT Department:

Programme Educational Objectives:

Graduates will be able to

Programme Outcomes (POs):

Engineering Graduates will be able to:

1. **Engineering knowledge**: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
2. **Problem analysis**: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. **Design/development of solutions**: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
4. **Conduct investigations of complex problems**: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. **Modern tool usage**: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
6. **The engineer and society**: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7. **Environment and sustainability**: Understand the impact of the professional engineering solutionsi n societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8. **Ethics**: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. **Individual and team work**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. **Communication**: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. **Project management and finance**: Demonstrate knowledge and understanding of the engineering and mana**gement principles and apply these to one's own work, as a member** and leader in a team, to manage projects and in multidisciplinary environments.
12. **Life-long learning**: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

SUBJECT INDEX

Introduction

A Road Map For OOA and OOD

In this subject, you have been hit with a bewildering array of new terminology. Words like "class", "instance", "method", "super class", "design", are all being thrown around with reckless abandon. It's easy to lose your place. The purpose of this road map is to help keep you grounded during the process of designing your object-oriented software systems.

Object-oriented program has become the dominant programming style in the software industry over the last 10 years or so. The reason for this has to do with the growing size and scale of software projects. It becomes extremely difficult to understand a procedural program once it gets above a certain size. Object-oriented programs scale up better, meaning that they are easier to write, understand and maintain than procedural programs of the same size. There are basically three reasons for this:

1. Object-oriented programs tend to be written in terms of real-world objects, not internal data structures. This makes them somewhat easier to understand by maintainers and the people who have to read your code -- but it may make it harder for you as the initial designer. Identifying objects in a problem is a challenge

2. Object-oriented programs encourage *encapsulation* -- details of an objects implementation are hidden from other objects. This keeps a change in one part of the program from affecting other parts, making the program easier to debug and maintain. This does take some getting used to -- past students have called it "hypertext programming" because your program is spread out among a variety of objects.

3. Object-oriented programs encourage modularity. This means that pieces of the program do not depend on other pieces of the program. Those pieces can be reused in future projects, making the new projects easier to build.

Because of the overhead costs referred to above, OOP is probably not the best choice for very small programs. If all you want to do is sum up file sizes in a directory, you probably don't want to have to identify objects, create reusable structures and all that. Since, the programs that you write for this class are necessarily small (so you can do them in the quarter), so it may not always be obvious where the big benefit from using objects is.

A good object-oriented program, then, is one that takes advantage of the strengths of object-oriented programs as listed above. Some top-level guidelines for doing this include:

- Objects in your system should usually represent real-world things. Talk about cash registers and vending machines and stop lights, not linked-lists, hash tables or binary trees.

- Objects in your system should have limited knowledge. They should only know about the other parts of the system that they absolutely have to in order to get their work done. This implies that you should *always* have more than one object in the system.

It's not always clear, however, how to implement goals like this in practice. That's where the design comes into play.

Why Design?

In order to get the most out of using objects, OO programs require some kind of design work before programming starts. You, as the programmer, are much more in the position of an architect. Why do architects spend so much time on blueprints and models? One reason is that the blueprint allows the architect to solve the problem in a relatively low-cost environment before actually using brick and concrete. The final building is better: safer, more energy-efficient, and so on, because the worst mistakes are corrected on paper, cheaply and quickly, before a building falls on somebody's head. As a nice side effect, the precision of the blueprint makes it a useful tool for communicating the design to the builders, saving time in the building process.

OO probably requires more up-front design work than procedural programs because the job of identifying the objects and figuring out how to divide the tasks among them is not really a programming job. You really do need to have that started before you sit down to hack -- otherwise, how will you know where to start?

Similarly, time spent getting the design right before you start programming will almost always save you time in the end. It's much, much easier to make major changes on a design, which is after all just squiggly lines on paper, and then it is to make changes in hundreds or thousands of lines of code.

Several Activities to a Brilliant Design

The design process is typically split into two distinct phases: Object-Oriented Analysis (OOA) and Object Oriented Design (OOD). Yes, I know that it is confusing to have one of the sub-steps of the design process also called "design". However, the term is so entrenched that it's hopeless to start creating new jargon.

We'll call these things "activities" rather than "steps" to emphasize that they don't have to be done in any particular order -- you can switch to another activity whenever it makes sense to do so. When you are actually doing this, you'll find that you want to go back and forth between OOA and OOD repeatedly. Maybe you'll start at one part of your program, and do OOA and OOD on that part before doing OOA and OOD on another part. Or maybe you'll do a preliminary OOA, and then decide while working on the OOD that you need more classes, so you jump back to OOA. That's great. Moving back and forth between OOA and OOD is the best way to create a good design -- if you only go through each step once; you'll be stuck with your first mistakes all the way through. I do think that it's important, though, to always be clear what activity you are currently doing -- keeping a sharp distinction between activities will make it easier for you to make design decisions without getting tied up in knots.

In the OOA phase, the overall questions is "What?". As in, "What will my program need to do?", "What will the classes in my program be?", and "What will each class be responsible for?” You do not worry about implementation details in the OOA phase -- there will be plenty of time to worry about them later, and at this point they only get in the way. The focus here is on the real world -- what are the objects, tasks and responsibilities of the real system?

In the OOD phase, the overall question is "How?” As in, "How will this class handle its responsibilities?", "How can I ensure that this class knows all the information it needs?", "How will classes in my design communicate?” At this point, you are worried about some implementation details, but not all -- what the attributes and methods of a class will be, but not down to the level of whether things are integers or reals or ordered collections or dictionaries or whatnot -- those are programming decisions.

1.SRS

Aim: Choose a hypothetical system of significant complexity and write an SRS for the same.

Theory:

Requirements Statement for Example ATM System

The software to be designed will control a simulated automated teller machine (ATM) having a magnetic stripe reader for reading an ATM card, a customer console (keyboard and display) for interaction with the customer, a slot for depositing envelopes, a dispenser for cash, a printer for printing customer receipts, and a key-operated switch to allow an operator to start or stop the machine. The ATM will communicate with the bank's computer over an appropriate communication link. (The software on the latter is not part of the requirements for this problem.)

The ATM will service one customer at a time. A customer will be required to insert an ATM card and enter a personal identification number (PIN) - both of which will be sent to the bank for validation as part of each transaction. The customer will then be able to perform one or more transactions. The card will be retained in the machine until the customer indicates that he/she desires no further transactions, at which point it will be returned - except as noted below.

The ATM must be able to provide the following services to the customer:

1. A customer must be able to make a cash withdrawal from any suitable account linked to the card. Approval must be obtained from the bank before cash is dispensed.

2. A customer must be able to make a deposit to any account linked to the card, consisting of cash and/or checks in an envelope. The customer will enter the amount of the deposit into the ATM, subject to manual verification when the envelope is removed from the machine by an operator. Approval must be obtained from the bank before physically accepting the envelope.

3. A customer must be able to make a transfer of money between any two accounts linked to the card.
4. A customer must be able to make a balance inquiry of any account linked to the card.

A customer must be able to abort a transaction in progress by pressing the Cancel key instead of responding to a request from the machine.

The ATM will communicate each transaction to the bank and obtain verification that it was allowed by the bank. Ordinarily, a transaction will be considered complete by the bank once it has been approved. In the case of a deposit, a second message will be sent to the bank indicating that the customer has deposited the envelope. (If the customer fails to deposit the envelope within the timeout period, or presses cancel instead, no second message will be sent to the bank and the deposit will not be credited to the customer.)

If the bank determines that the customer's PIN is invalid, the customer will be required to re-enter the PIN before a transaction can proceed. If the customer is unable to successfully enter the PIN after three tries, the card will be permanently retained by the machine, and the customer will have to contact the bank to get it back.

If a transaction fails for any reason other than an invalid PIN, the ATM will display an explanation of the problem, and will then ask the customer whether he/she wants to do another transaction.

The ATM will provide the customer with a printed receipt for each successful transaction, showing the date, time, machine location, type of transaction, account(s), amount, and ending and available balance(s) of the affected account ("to" account for transfers).

The ATM will have a key-operated switch that will allow an operator to start and stop the servicing of customers. After turning the switch to the "on" position, the operator will be required to verify and enter the total cash on hand. The machine can only be turned off when it is not servicing a customer. When the switch is moved to the "off" position, the machine will shut down, so that the operator may remove deposit envelopes and reload the machine with cash, blank receipts, etc.

The ATM will also maintain an internal log of transactions to facilitate resolving ambiguities arising from a hardware failure in the middle of a transaction. Entries will be made in the log when the ATM is started up and shut down, for each message sent to the Bank (along with the response back, if one is expected), for the dispensing of cash, and for the receiving of an envelope. Log entries may contain card numbers and dollar amounts, but for security will *never* contain a PIN.

2. USE CASE DIAGRAM.

Aim: Draw one or more Use Case diagrams for capturing and representing requirements of the system. Use case diagrams must include template showing description and steps of the Use Case for various scenarios.

Theory: A use case diagram establishes the capability of the system as a whole.

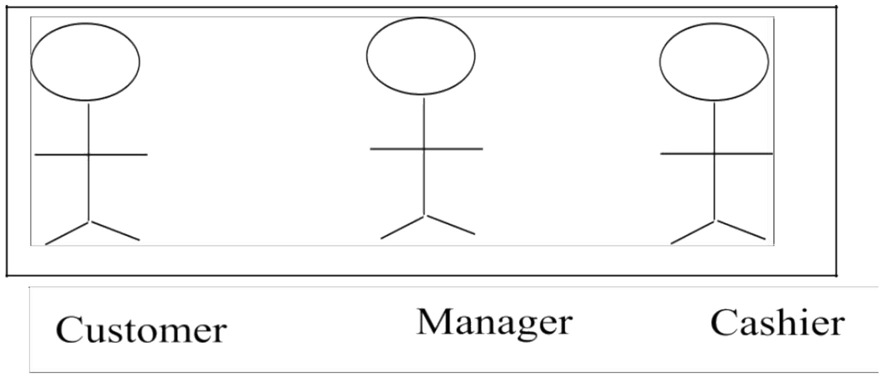
Components of use case diagram:

Actor, Use case, System boundary, Relationship, Actor relationship
Semantic of the components is followed.

- What is an actor?

  An actor is someone or something that must interact with the system under development

  UML notation for actor is stickman, shown below



- 4-Categories of an actor:

Principle: Who uses the main system functions.

Secondary: Who takes care of administration & maintenance.

External h/w: The h/w devices which are part of application domain and must be used.

Other system: The other system with which the system must interact.

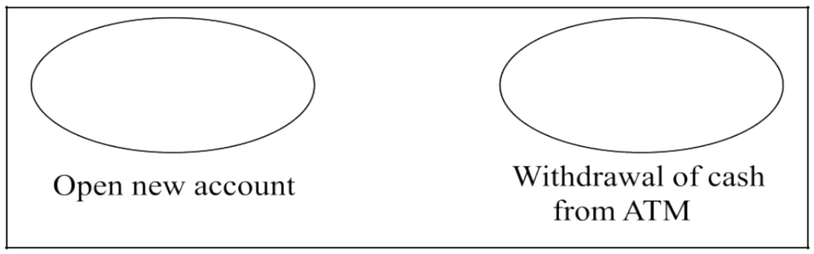
- What is USE case?

  A use case is a pattern of behavior, the system exhibits

  Each use case is a sequence of related transactions performed by an actor and the system in dialogue.

  USE CASE is dialogue between an actor and the system.

  Examples:



- **Generic format for documenting the use case:**

 - Pre condition:If any

- Use case: Name of the case.
- Actors: List of actors(external agents), indicating who initiates the use case.
- Purpose: Intention of the use case.
- Overview: Description.
- Type:primary / secondary.
- Post condition:If any

Typical Course of Events:

ACTOR ACTION: Numbered actions of the actor.

SYSTEM RESPONSE: Numbered description of system responses.

- What is System Boundary?

- It is shown as a rectangle.
- It helps to identify what is an external verse internal, and what the responsibilities of the system are.
- The external environment is represented only by actors.

- What is Relationship?

  Relationship between use case and actor: Communicates

  Relationship between two use cases: Extends, Uses

  Notation used to show the relationships: << >>

Relationship between use case and actor is often referred as "communicates" Relationship between two use cases is refereed as either uses or extends.

USES:

- Multiple use cases share a piece of same functionality.
- This functionality is placed in a separate use case rather than documenting in every use case that needs it.

A uses relationship shows behavior that is common to one or more use cases.

EXTENDS:

It is used to show optional behavior, which is required only under certain condition

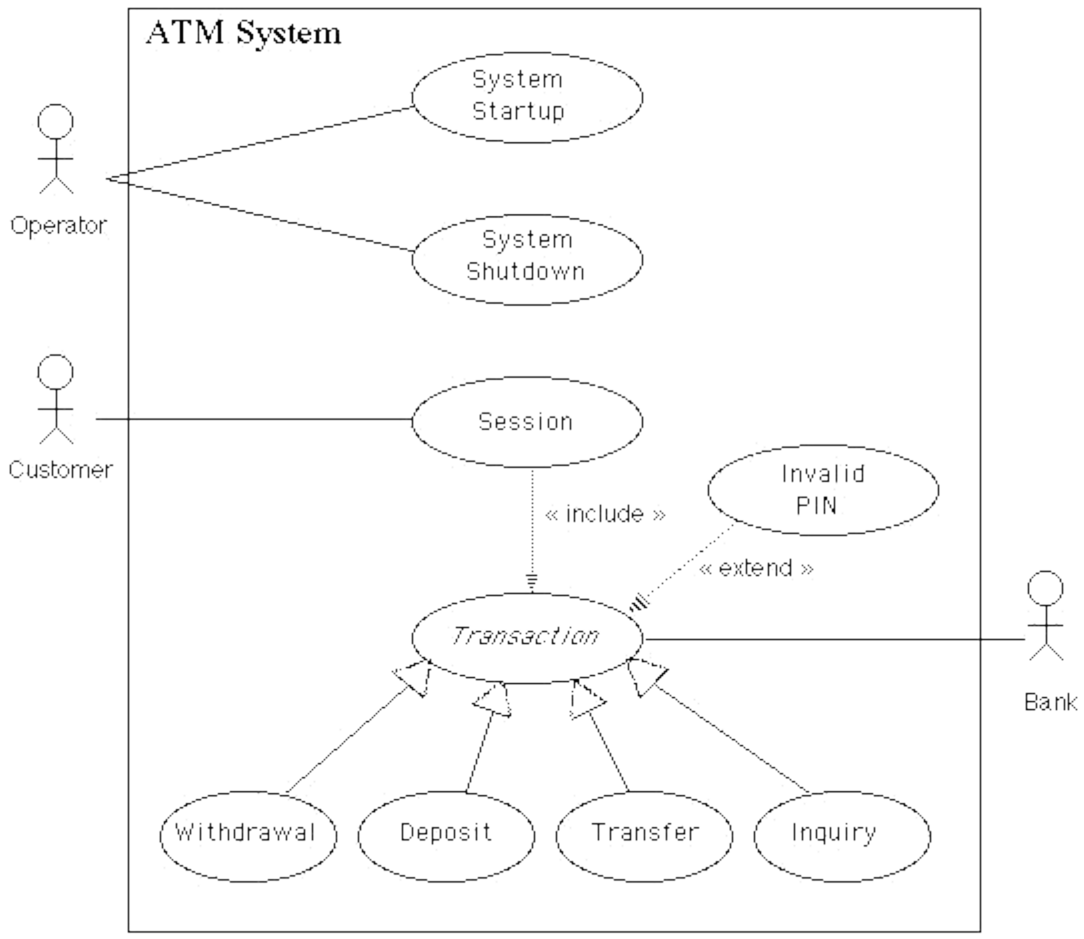
Use Cases for ATM System

System Startup Use Case

The system is started up when the operator turns the operator switch to the "on" position. The operator will be asked to enter the amount of money currently in the cash dispenser, and a connection to the bank will be established. Then the servicing of customers can begin.

System Shutdown Use Case

The system is shut down when the operator makes sure that no customer is using the machine, and then turns the operator switch to the "off" position. The connection to the bank will be shut down. Then the operator is free to remove deposited envelopes, replenish cash and paper, etc.



<u>Session Use Case</u>

A session is started when a customer inserts an ATM card into the card reader slot of the machine. The ATM pulls the card into the machine and reads it. (If the reader cannot read the card due to improper insertion or a damaged stripe, the card is ejected, an error screen is displayed, and the session is aborted.) The customer is asked to enter his/her PIN, and is then allowed to perform one or more transactions, choosing from a menu of possible types of transaction in each case. After each transaction, the customer is asked whether he/she would like to perform another. When the customer is through performing transactions, the card is ejected from the machine and the session ends. If a transaction is aborted due to too many invalid PIN entries, the session is also aborted, with the card being retained in the machine.

The customer may abort the session by pressing the Cancel key when entering a PIN or choosing a transaction type.

Transaction Use Case

A transaction use case is started within a session when the customer chooses a transaction type from a menu of options. The customer will be asked to furnish appropriate details (e.g. account(s) involved, amount). The transaction will then be sent to the bank, along with information from the customer's card and the PIN the customer entered.

If the bank approves the transaction, any steps needed to complete the transaction (e.g. dispensing cash or accepting an envelope) will be performed, and then a receipt will be printed. Then the customer will be asked whether he/she wishes to do another transaction.

If the bank reports that the customer's PIN is invalid, the Invalid PIN extension will be performed and then an attempt will be made to continue the transaction. If the customer's card is retained due to too many invalid PINs, the transaction will be aborted, and the customer will not be offered the option of doing another.

If a transaction is cancelled by the customer, or fails for any reason other than repeated entries of an invalid PIN, a screen will be displayed informing the customer of the reason for the failure of the transaction, and then the customer will be offered the opportunity to do another.

The customer may cancel a transaction by pressing the Cancel key as described for each individual type of transaction below.

All messages to the bank and responses back are recorded in the ATM's log.

Withdrawal Transaction Use Case

A withdrawal transaction asks the customer to choose a type of account to withdraw from (e.g. checking) from a menu of possible accounts, and to choose a dollar amount from a menu of possible amounts. The system verifies that it has sufficient money on hand to satisfy the request before sending the transaction to the bank. (If not, the customer is informed and asked to enter a different amount.) If the transaction is approved by the bank, the appropriate amount of cash is dispensed by the machine before it issues a receipt. (The dispensing of cash is also recorded in the ATM's log.)

A withdrawal transaction can be cancelled by the customer pressing the Cancel key any time prior to choosing the dollar amount.

Deposit Transaction Use Case

A deposit transaction asks the customer to choose a type of account to deposit to (e.g. checking) from a menu of possible accounts, and to type in a dollar amount on the keyboard.

The transaction is initially sent to the bank to verify that the ATM can accept a deposit from this customer to this account. If the transaction is approved, the machine accepts an envelope from the customer containing cash and/or checks before it issues a receipt. Once the envelope has been received, a second message is sent to the bank, to confirm that the bank can credit the customer's account - contingent on manual verification of the deposit envelope contents by an operator later. (The receipt of an envelope is also recorded in the ATM's log.)

A deposit transaction can be cancelled by the customer pressing the Cancel key any time prior to inserting the envelope containing the deposit. The transaction is automatically cancelled if the customer fails to insert the envelope containing the deposit within a reasonable period of time after being asked to do so.

Transfer Transaction Use Case

A transfer transaction asks the customer to choose a type of account to transfer from (e.g. checking) from a menu of possible accounts, to choose a different account to transfer to, and to type in a dollar amount on the keyboard. No further action is required once the transaction is approved by the bank before printing the receipt.

A transfer transaction can be cancelled by the customer pressing the Cancel key any time prior to entering a dollar amount.

Inquiry Transaction Use Case

An inquiry transaction asks the customer to choose a type of account to inquire about from a menu of possible accounts. No further action is required once the transaction is approved by the bank before printing the receipt.

An inquiry transaction can be cancelled by the customer pressing the Cancel key any time prior to choosing the account to inquire about.

Invalid PIN Extension

An invalid PIN extension is started from within a transaction when the bank reports that the customer's transaction is disapproved due to an invalid PIN. The customer is required to re-enter the PIN and the original request is sent to the bank again. If the bank now approves the transaction, or disapproves it for some other reason, the original use case is continued; otherwise the process of re-entering the PIN is repeated. Once the PIN is successfully re-entered, it is used for both the current transaction and all subsequent transactions in the session. If the customer fails three times to enter the correct PIN, the card is permanently retained, a screen is displayed informing the customer of this and suggesting he/she contact the bank, and the entire customer session is aborted. If the customer presses cancel instead of re-entering a PIN, the original transaction is cancelled.

3. CLASS DIAGRAM

Aim: Draw basic class diagrams to identify and describe key concepts like classes, types in your system and their relationships.

Theory:

A class diagram shows the existence of classes and their relationships in the logical view of a system

UML modeling elements in class diagrams are:

– Classes, their structure and behavior.

– relationships components among the classes like association, aggregation, composition, dependency and inheritance

– Multiplicity and navigation indicators

– Role names or labels.

➢ Major Types of classes:

Concrete classes

- A concrete class is a class that is instantiable; that is it can have different instances.
- Only concrete classes may be leaf classes in the inheritance tree.

Abstract classes

- An abstract class is a class that has no direct instance but whose **descendant's** classes have direct instances.
- An abstract class can define the protocol for an operation without supplying a corresponding method we call this as an *abstract operation.*
- An abstract operation defines the form of operation, for which each concrete subclass should provide its own implementation.

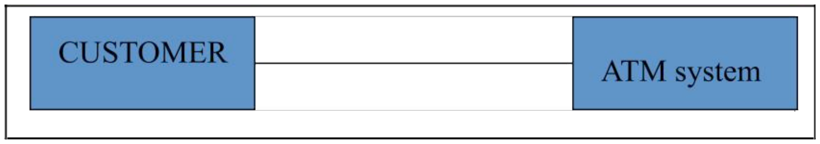
➢ RELATIONSHIP:

- Association
- Aggregation
- Composition
- Inheritance
- Dependency
- Instantiation

ASSOCIATION:

These are the most general type of relationship:

- It denotes a semantic connection between two classes
- It shows BI directional connection between two classes. It is a weak coupling as associated classes remain somewhat independent of each other

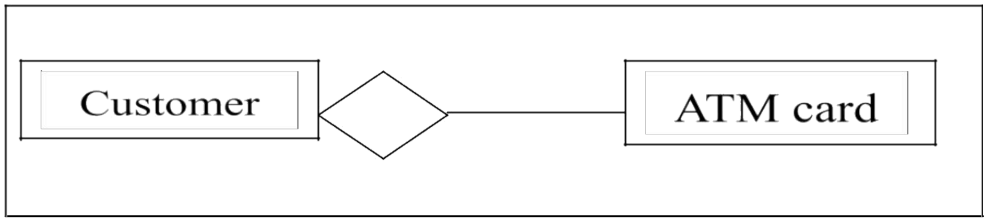


AGGREGATION:

This is a special type of association

- The association with label “contains” or “is part of” is an aggregation
- It represents “has a “ relationship
- It is used when one object logically or physically contains other
- The container is called as aggregate
- It has a diamond at its end
- The components of aggregate can be shared with others
- It expresses a whole - part relationships

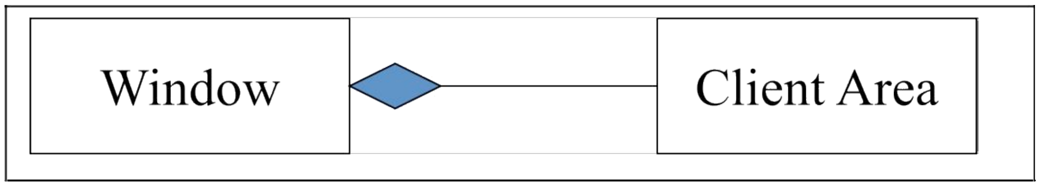
Example:



COMPOSITION:

This is a strong form of aggregation

- It expresses the stronger coupling between the classes
- The owner is explicitly responsible for creation and deletion of the part
- Any deletion of whole is considered to cascade its part
- The aggregate has a filled diamond at its end



INHERITANCE:

- The inheritance relationship helps in managing the complexity by ordering objects within trees of classes with increasing levels of abstraction. Notation used is solid line with arrow head, shown below.

- Generalization and specialization are points of view that are based on inheritance hierarchies.

DEPENDENCY:

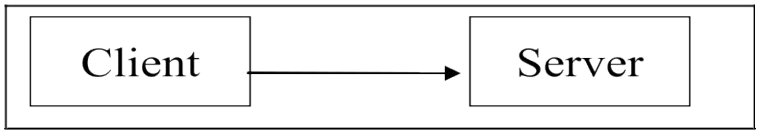
- Dependency is semantic connection between dependent and independent model elements.

- This association is unidirectional and is shown with dotted arrowhead line.

- In the following example it shows the dependency relationship between client and server.

- The client avails services provided by server so it should have semantic knowledge of server.

The server need not know about client.

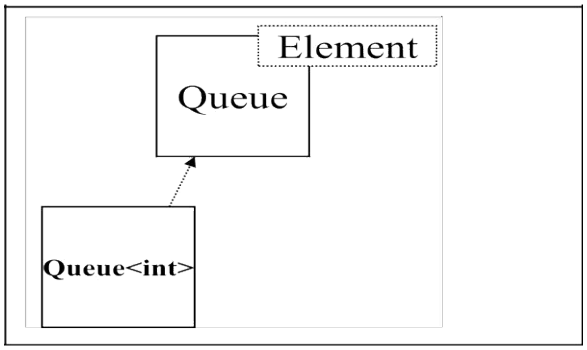


INSTANTIATION

This relationship is defined between parameterized class and actual class.

- Parameterized class is also referred as generic class.
- **A parameterized class can't have instances unless we first instantiated it**

Example:



<u>Class Diagram for ATM System</u>

<u>Class diagram for withdrawal</u>

Withdrawal

-

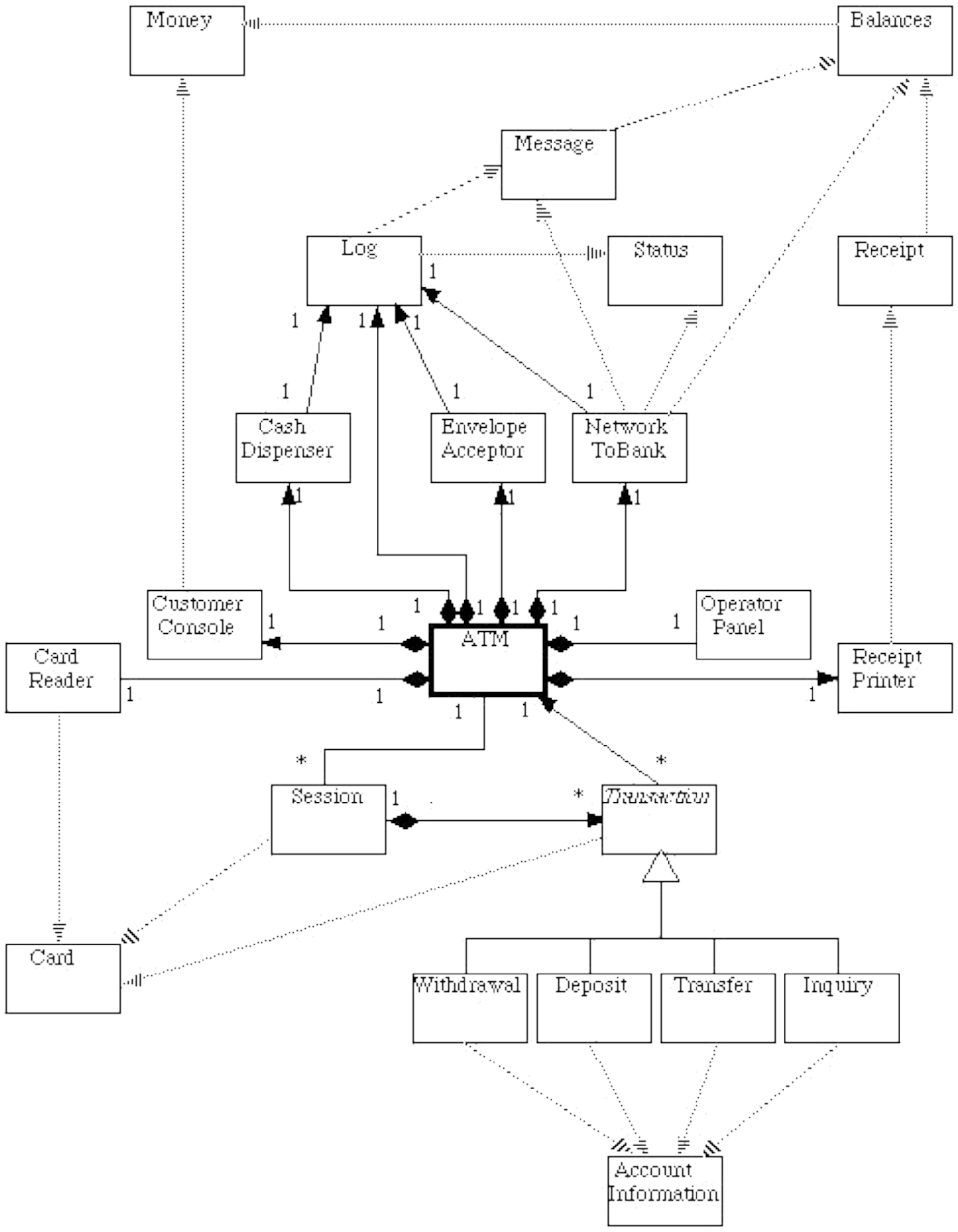
- from: int

- amount: Money

+ <u>Withdrawal(atm: ATM, session: Session, card: Card, pin: int)</u>

# getSpecificsFromCustomer(): Message throws Cancelled

# completeTransaction(): Receipt


Money
Balances
Message
Log
Status
Receipt
Cash Dispenser
Envelope Acceptor
Network ToBank
Customer Console
Operator Panel
Card Reader
ATM
Receipt Printer
Session
Transaction
Card
Withdrawal
Deposit
Transfer
Inquiry
Account Information
1
*

CRC Cards for ATM

Using CRC cards to assign responsibilities to various classes for the tasks required by the various use cases leads to the creation of the following cards.

1. Class ATM

Responsibilities	**Collaborators**
Start up when switch is turned on	OperatorPanel CashDispenser
Shut down when switch is turned off	NetworkToBank
Start a new session when card is inserted by customer	[redacted] CustomerConsole Session
Provide access to component parts for sessions and transactions	NetworkToBank

2. Class CardReader

[redacted]

Responsibilities	**Collaborators**
[redacted]	
Tell ATM when card is inserted	ATM
Read information from card	Card
Eject card	
Retain card	

3. Class CustomerConsole

Responsibilities	**Collaborators**
Display a message	
Display a prompt, accept a PIN from keyboard	
Display a prompt and menu, accept a choice from keyboard	
Display a prompt, accept a dollar amount from keyboardRespond to cancel key being pressed by customer	

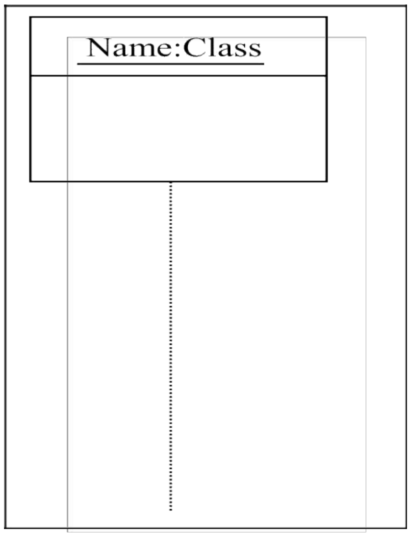
*4.*Sequence Diagram

Aim: Draw sequence diagrams OR communication diagrams with advanced notation for your system to show objects and their message exchanges.

Theory:

- Typically these diagrams capture behaviors of the single scenario.
- Shows object interaction arranged in time sequence.
- They show sequence of messages among the objects.
- It has two dimensions, vertical represents time & horizontal represents objects.
- **Components of sequence diagram:**
    - objects
    - object lifeline
    - Message
    - pre/post conditions.

OBJECT & OBJECT LIFE LINE



- Object are represented by rectangles and name of the objects are underlined.
- Object life line are denoted as dashed lines. They are used to model the existence of objects over time.

MESSAGES:

- They are used to model the content of communication between objects. They are used to convey information between objects and enable objects to request services of other objects.

- The message instance has a sender, receiver, and possibly

  other information according to the characteristics of the request.

- Messages are denoted as labeled horizontal arrows between life lines.

- The sender will send the message and receiver will receive the message.

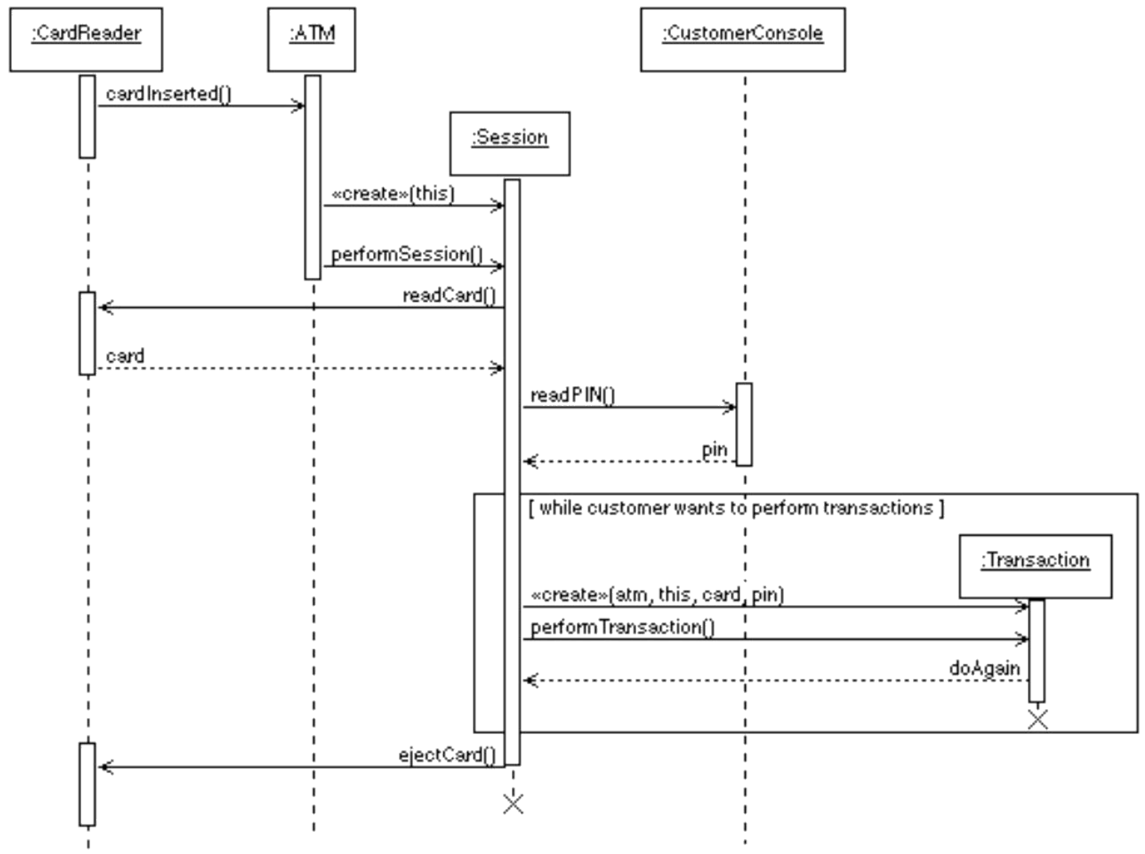
- May have square brackets containing a guard conditions. This is a Boolean condition that must be satisfied to enable the message to be sent.

- May have an asterisk followed by square brackets containing an iteration specification. This specifies the number of times the message is sent.

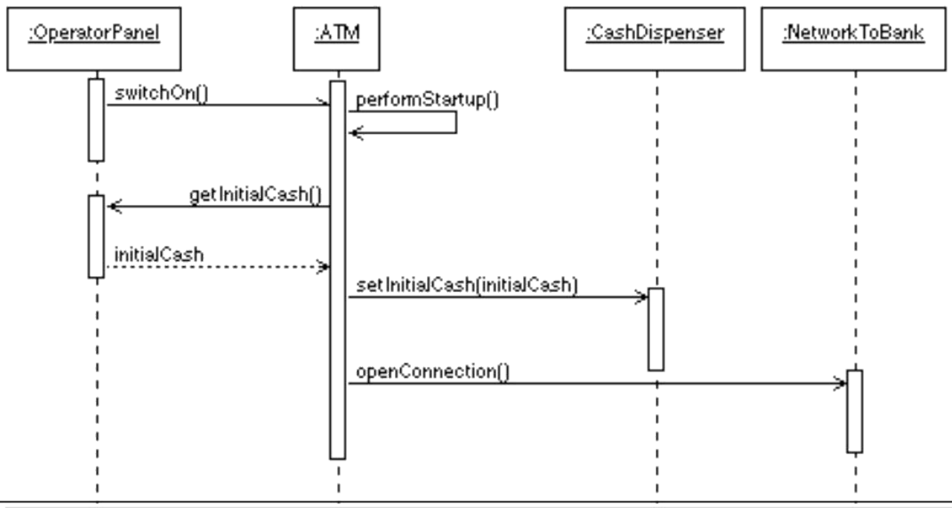
- May have return list consisting of a comma -separated list of names that designate the values of returned by the operation.

- Must have a name or identifier string that represents the message.

Session Sequence Diagram



System Startup Sequence Diagram

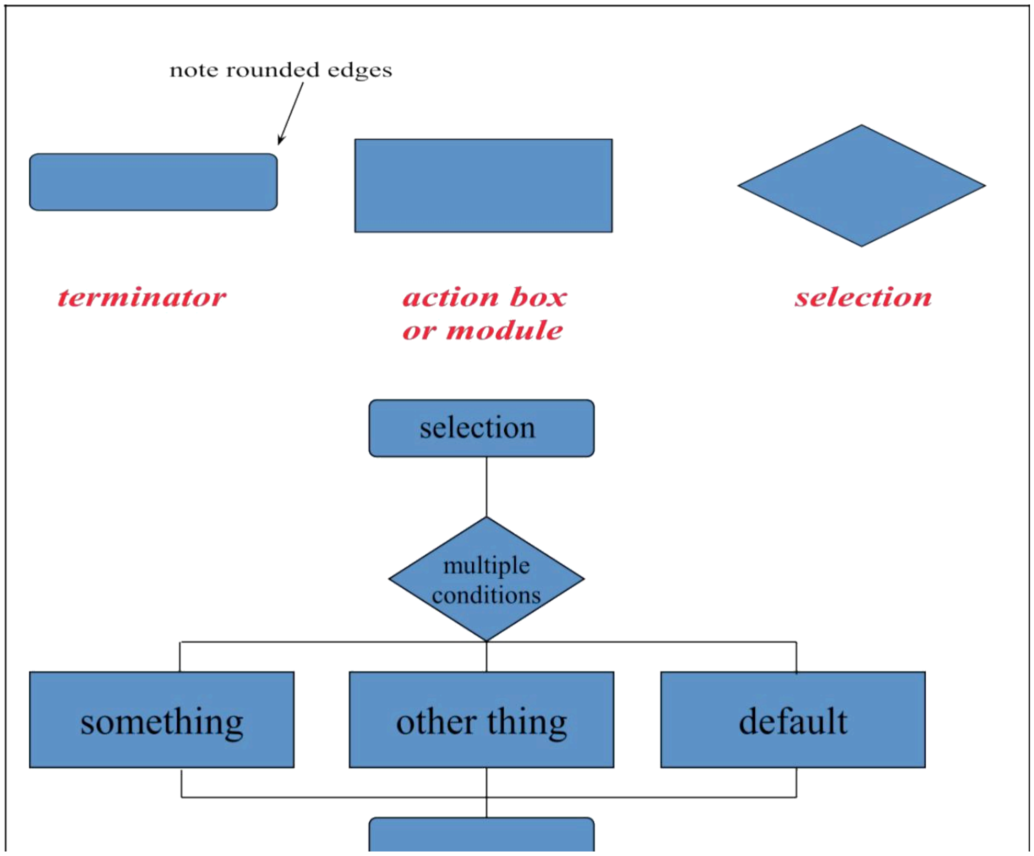


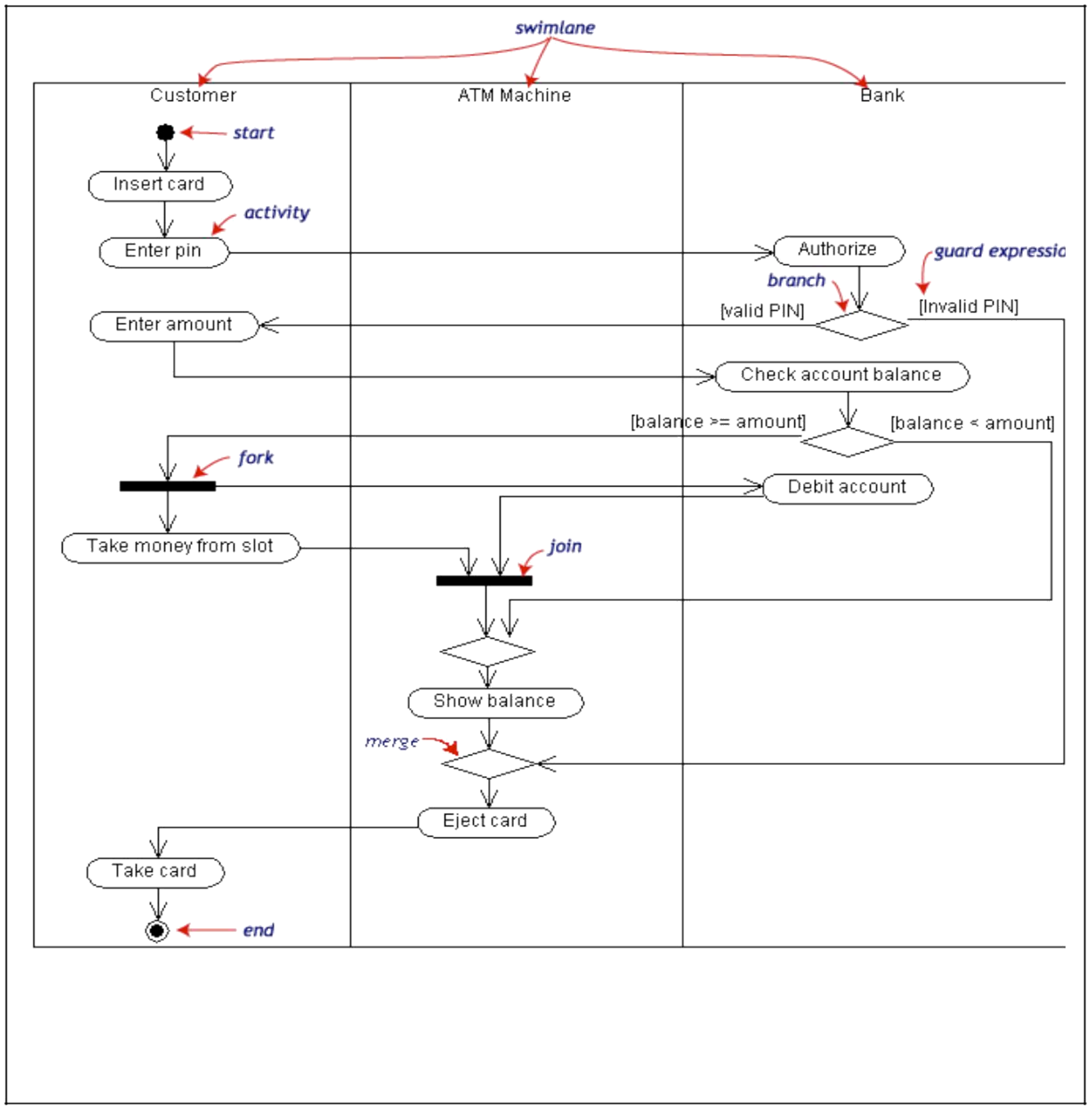
5.Activity Diagram

Aim: Draw activity diagrams to display either business flows or like flow charts.

Theory:

- It is a special kind of state diagram and is worked out at use case level.
- These are mainly targeted towards representing internal behavior of a use case.
- These may be thought as a kind of flowchart.
- Flowcharts are normally limited to sequential process; activity diagrams can handle parallel process.
- Activity diagrams are recommended in the following situations:
    - Analyzing use case
    - Dealing with multithreaded application
    - Understanding workflow across many use cases.
- activity diagram symbols





ACTIVITY DIAGRAM

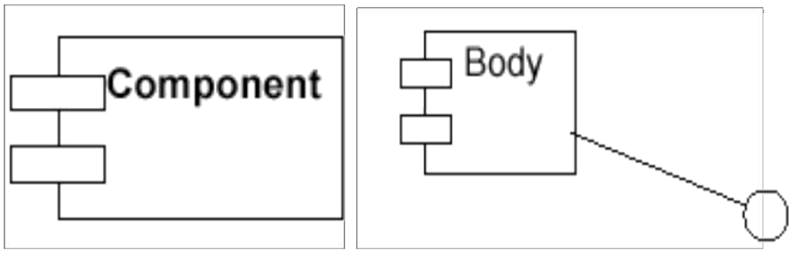
6. Component Diagram

Aim: Draw component diagrams assuming that you will build your system reusing existing components along with a few new ones.

Theory:

Component Diagram Notation

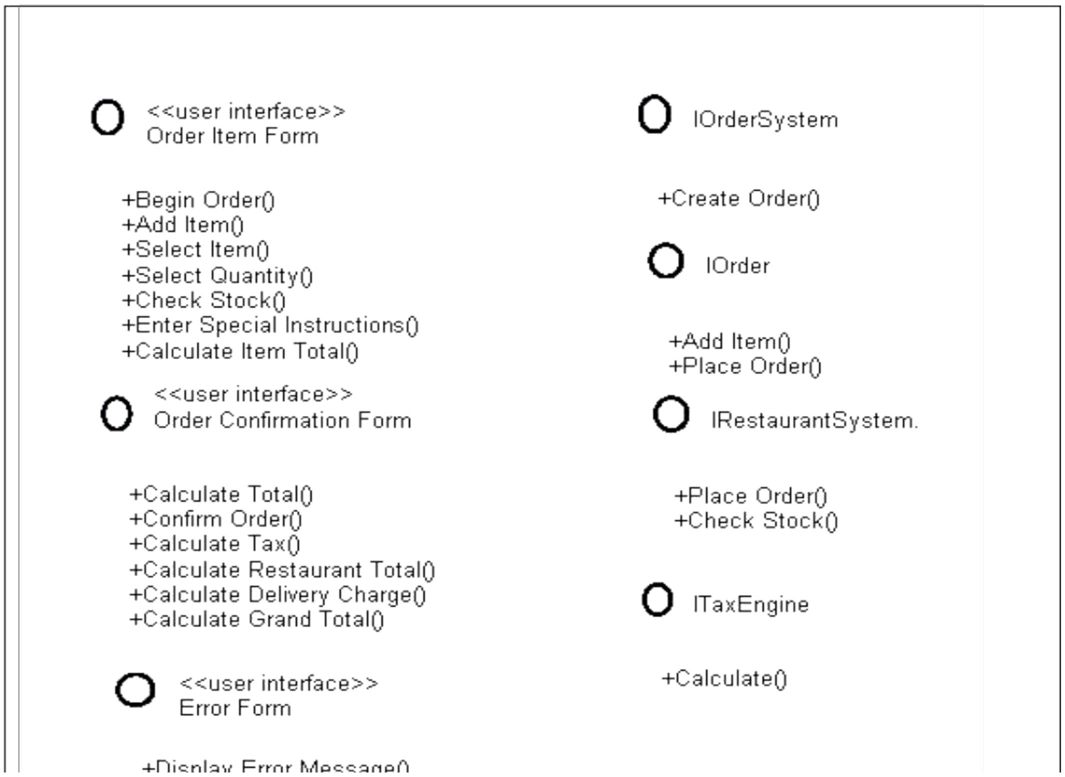
- Components are shown as rectangles with two tabs at the upper left
- Dashed arrows indicate dependencies
- Circle and solid line indicates an interface to the component



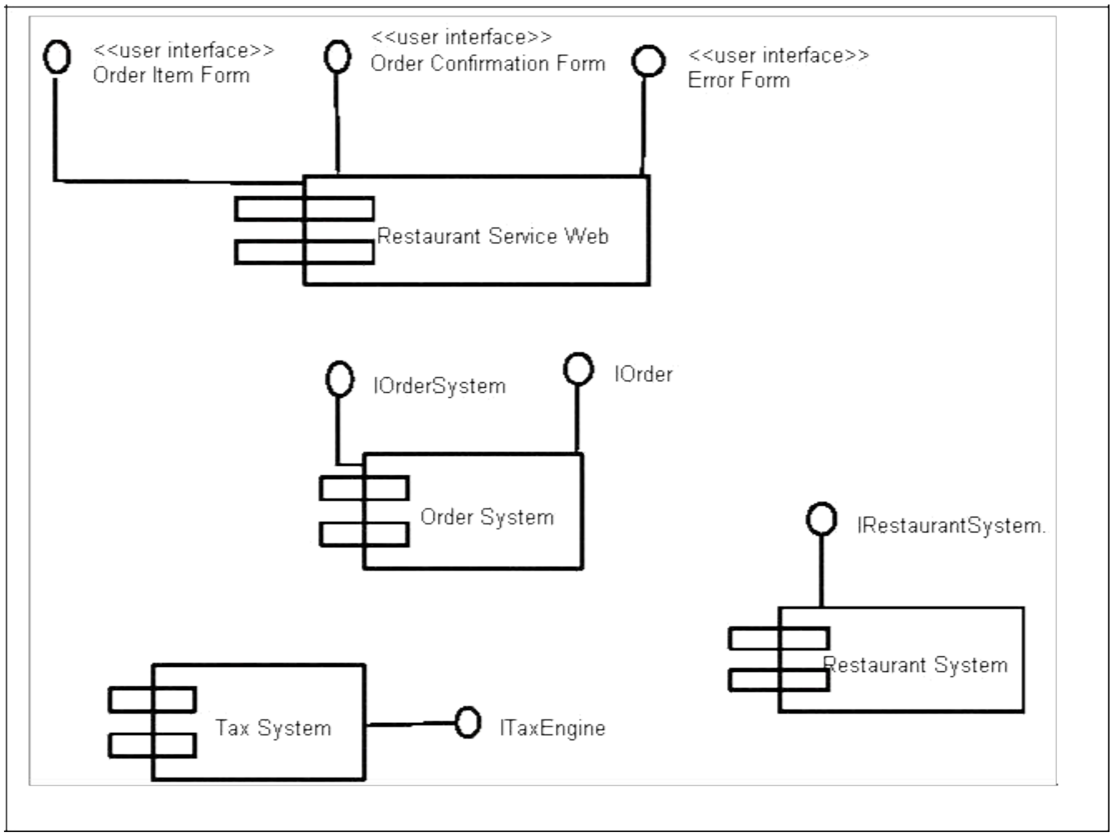
Component

Example - Interfaces

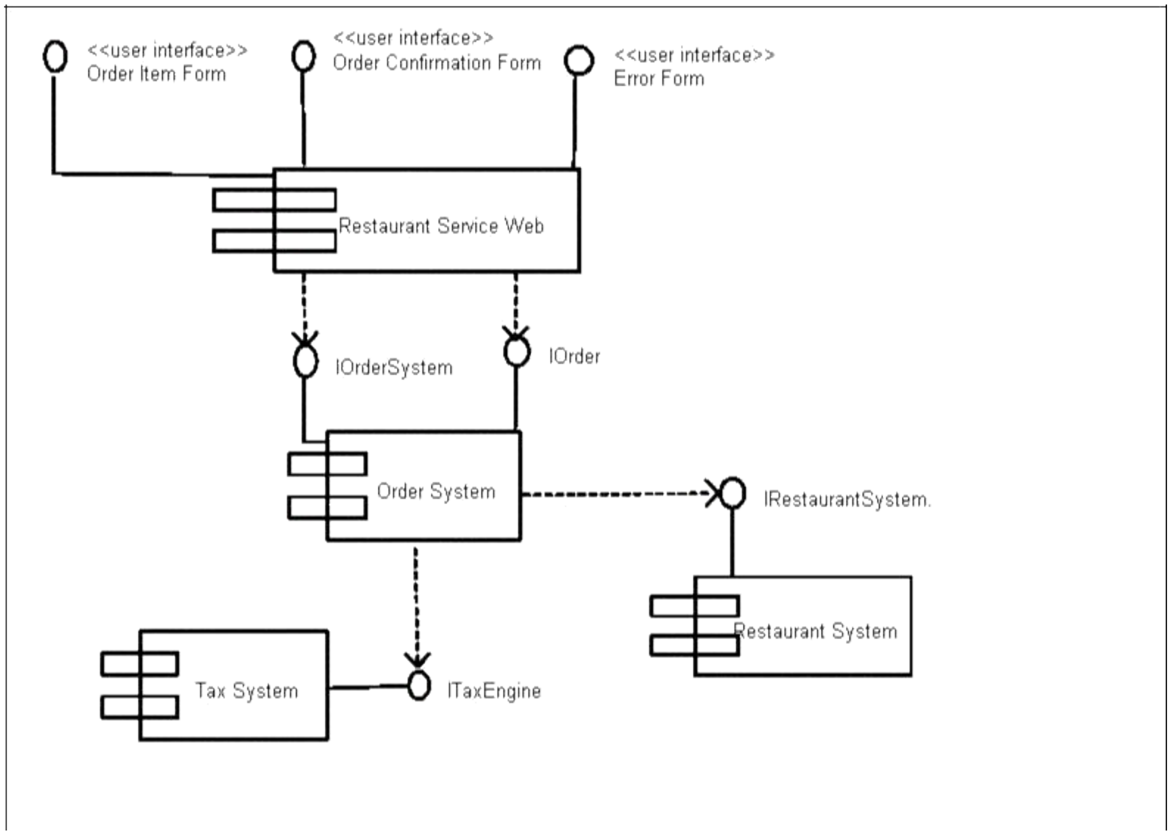
- Restaurant ordering system
- Define interfaces first – comes from Class Diagrams



Graphical depiction of components



Component Example - Linking



7. Deployment Diagram

Aim: Draw deployment diagrams to model the runtime architecture of your system.

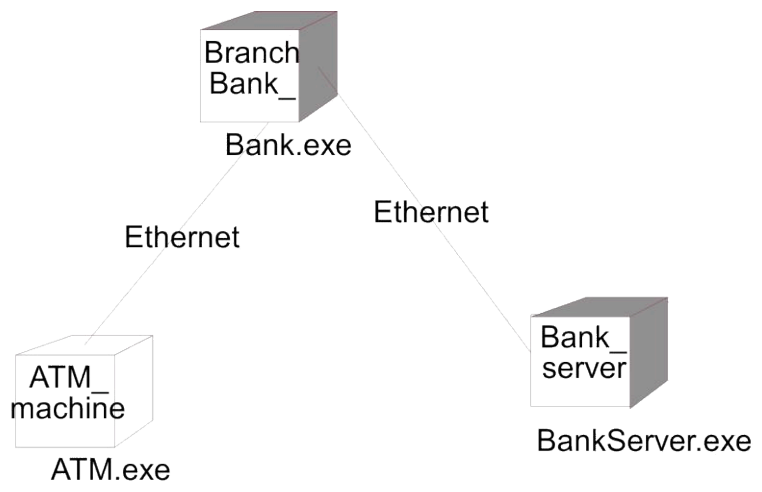
Theory:

- Shows the physical architecture of the hardware and software of the deployed system
- Nodes
    - Typically contain components or packages
    - Usually some kind of computational unit; e.g. machine or device (physical or logical)
- Physical relationships among software and hardware in a delivered systems
    - Explains how a system interacts with the external environment

Deployment Diagram for ATM system

- A deployment diagram shows the relationship among software and hardware components in the delivered system.
- These diagrams include nodes and connections between nodes.
- Each node in deployment diagram represents some kind of computational unit, in most cases a piece of hardware.
- Connection among nodes show the communication path over which the system will interact.

- The connections may represent direct hardware coupling line RS-232 cable, Ethernet connection, they also may represent indirect coupling such as satellite to ground communication**.**



DESIGN PATTERN

INTRODUCTION

Design patterns represent the best practices used by experienced object-oriented software developers. Design patterns are solutions to general problems that software developers faced during software development. These solutions were obtained by trial and error by numerous software developers over quite a substantial period of time.

Gang of Four (GOF)

In 1994, four authors Erich Gamma, Richard Helm; Ralph Johnson und John Vlissides published a book titled Design Patterns - Elements of Reusable Object-Oriented Software which initiated the concept of Design Pattern in Software development. These authors are collectively known as Gang of Four (GOF). According to these authors design patterns are primarily based on the following principles of object orientated design.

- Program to an interface not an implementation
- Favor object composition over inheritance

Usage of Design Pattern

Design Patterns have two main usages in software development.

Design patterns provide a standard terminology and are specific to particular scenario. For example, a singleton design pattern signifies use of single object so all developers familiar with single design pattern will make use of single object and they can tell each other that program is following a singleton pattern. Design patterns have been evolved over a long period of time and they provide best solutions to certain problems faced during software development. Learning these patterns helps inexperienced developers to learn software design in an easy and faster way.

Types of Design Pattern

As per the design pattern reference book Design Patterns - Elements of Reusable Object Oriented Software, there are 23 design patterns. These patterns can be classified in three categories: Creational, Structural and behavioral patterns. We'll also discuss another category of design patterns: J2EE design patterns.

S.N.	Pattern & Description
1	**Creational Patterns** These design patterns provides 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.
2	**Structural Patterns** These design patterns concern class and object composition. Concept of inheritance is used to compose interfaces and define ways to compose objects to obtain new functionalities.
3	**Behavioral Patterns** These design patterns are specifically concerned with communication between objects.
4	**J2EE Patterns** These design patterns are specifically concerned with the presentation tier. These patterns are identified by Sun Java Center.

8. ABSTRACT FACTORY PATTERN

Aim: Implement Abstract Factory Pattern using Java

Theory:

Abstract Factory patterns works around a super-factory which creates other factories. This factory is also called as Factory of factories. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object. In Abstract Factory pattern an interface is responsible for creating a factory of related objects,without explicitly specifying their classes. Each generated factory can give the objects as per the Factory pattern.

Implementation

We're going to create a *Shape* and *Color* interfaces and concrete classes implementing these interfaces. We creates an abstract factory class *AbstractFactory* as next step. Factory classes*ShapeFactory* and *ColorFactory* are defined where each factory extends *AbstractFactory*. A factory creator/generator class *FactoryProducer* is created.

AbstractFactoryPatternDemo, our demo class uses *FactoryProducer* to get

a *AbstractFactory* object. It will pass information (*CIRCLE / RECTANGLE / SQUARE* for *Shape*)to *AbstractFactory* to get the type of object it needs. It also passes information (*RED / GREEN /BLUE* for *Color*) to *AbstractFactory* to get the type of object it needs.

Steps

Use the following steps to implement the above mentioned design pattern.

Step 1

Create an interface for Shapes.

Shape.java

Step 2

Create concrete classes implementing the same interface.

Rectangle.java

Square.java

Step 3

Create an interface for Colors.

Color.java

Step4

Green.java

Blue.java

Step 5

Create an Abstract class to get factories for Color and Shape Objects.

AbstractFactory.java

Step 6

Create Factory classes extending AbstractFactory to generate object of concrete class based on

given information.

ShapeFactory.java

ColorFactory.java

}

Step 7

Create a Factory generator/producer class to get factories by passing an information such as Shape or Color

FactoryProducer.java

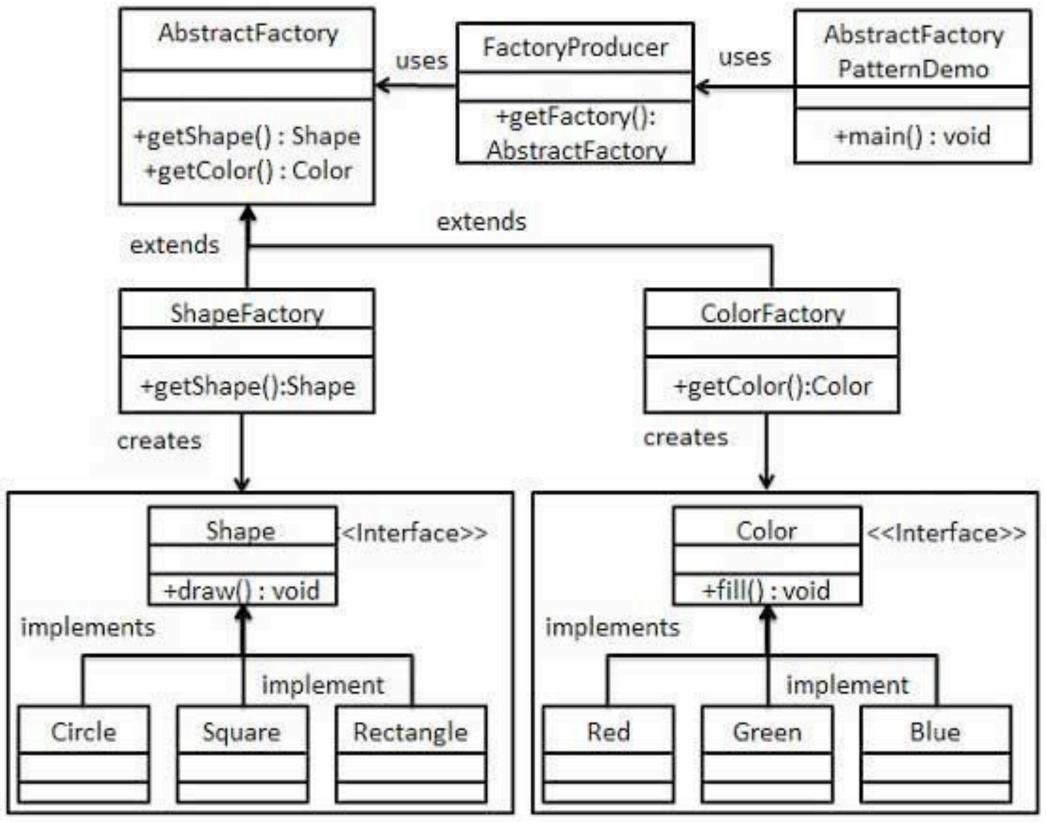
Step 8

Use the FactoryProducer to get AbstractFactory in order to get factories of concrete classes by

passing information such as type.

AbstractFactoryPatternDemo.java

Class Diagram



Step 9

Verify the output.

```
Inside Circle::draw() method.

Inside Rectangle::draw() method.

Inside Square::draw() method.

Inside Red::fill() method.Inside Green::fill() method.

Inside Blue::fill() method.
```

9. SIGLETON DESIGN PATTERN

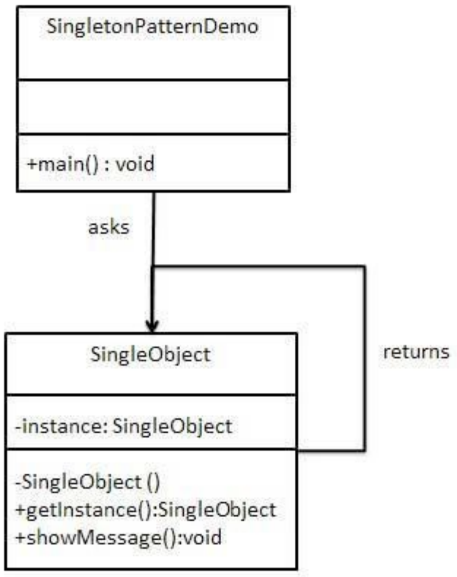
Singleton pattern is one of the simplest design patterns in Java. This type of design pattern comes under creational pattern as this pattern provides one of the best way to create an object. This pattern involves a single class which is responsible to creates own object while making sure that only single object get created. This class provides a way to access its only object which can be accessed directly without need to instantiate the object of the class.

Implementation

We're going to create a *SingleObject* class. *SingleObject* class have its constructor as private and have a static instance of itself.*SingleObject* class provides a static method to get its static instance to outside

world.*SingletonPatternDemo*, our demo class will use *SingleObject* class to geta *SingleObject* object.

Class Diagram



Steps

Use the following steps to implement the above mentioned design pattern.

Step 1

Create a Singleton Class.

SingleObject.java

Step 2

Get the only object from the singleton class.

SingletonPatternDemo.java

Step 3

Verify the output.

Hello World!

10. DECORRATOR DESIGN PATTERN

Decorator pattern allows adding new functionality an existing object without altering its structure. This type of design pattern comes under structural pattern as this pattern acts as a wrapper to existing class. This pattern creates a decorator class which wraps the original class and provides additionalfunctionality keeping class methods signature intact.

We are demonstrating use of Decorator pattern via following example in which we'll decorate a shape with some color without alter shape class.

Implementation

We're going to create a *Shape* interface and concrete classes implementing the *Shape* interface.We then create a abstract decorator class *ShapeDecorator* implementing the *Shape* interface and having*Shape* object as its instance variable.

RedShapeDecorator is concrete class implementing *ShapeDecorator*.

DecoratorPatternDemo, our demo class will use *RedShapeDecorator* to decorate *Shape* objects.

Steps

Use the following steps to implement the above mentioned design pattern.

Step 1

Create an interface.

Shape.java

Step 2

Create concrete classes implementing the same interface.

Rectangle.java

Step 3

Create abstract decorator class implementing the *Shape* interface.

ShapeDecorator.java

Step 4

Create concrete decorator class extending the *ShapeDecorator* class.

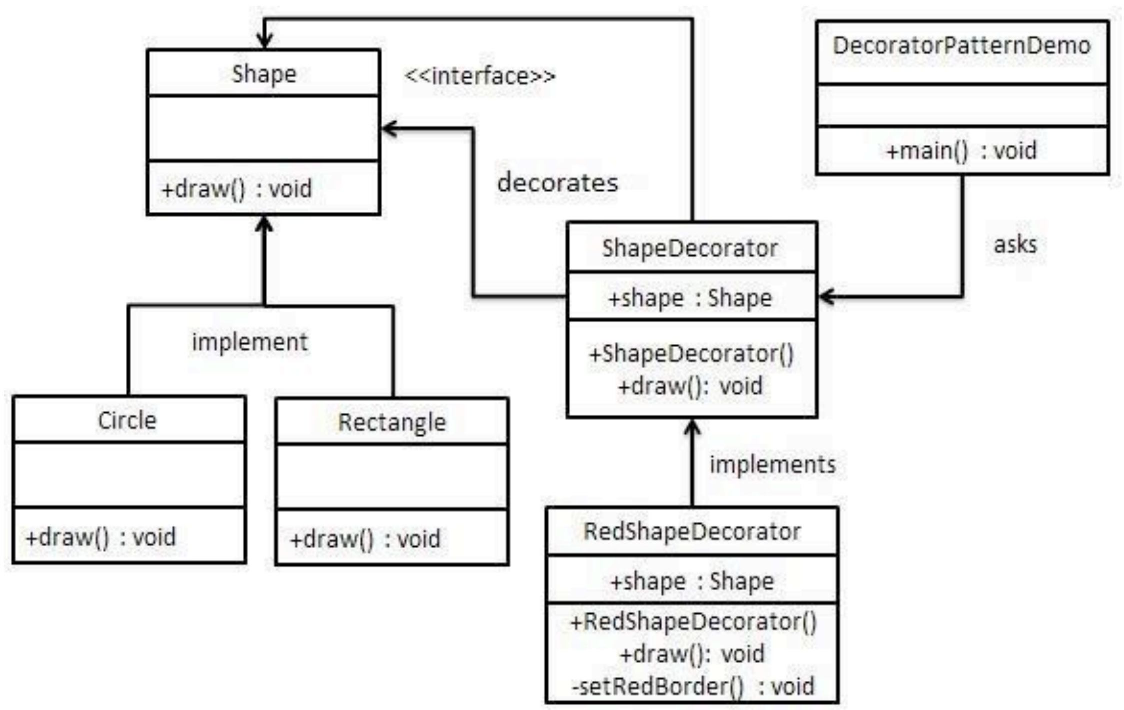
RedShapeDecorator.java

Step 5

Use the *RedShapeDecorator* to decorate *Shape* objects.

DecoratorPatternDemo.java

Class Diagram



Step 6

Verify the output.

```
Circle with normal border
Shape: Circle

Circle of red border
Shape: Circle
Border Color: Red

Rectangle of red border
Shape: Rectangle
Border Color: Red
```

11. ADAPTER DESIGN PATTERN

Adapter pattern works as a bridge between two incompatible interfaces. This type of design pattern comes under structural pattern as this pattern combines the capability of two independent interfaces. This pattern involves a single class which is responsible to join functionalities of independent or incompatible interfaces.

A real life example could be a case of card reader which acts as an adapter between memory card and a laptop. You plugins the memory card into card reader and card reader into the laptop so that memory card can be read via laptop. We are demonstrating use of Adapter pattern via 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.

Implementation

We've an interface *MediaPlayer* interface and a concrete class *AudioPlayer* implementing the*MediaPlayer* interface. *AudioPlayer* can play mp3 format audio files by default.We're

having another interface *AdvancedMediaPlayer* and concrete classes implementingthe*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've 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 the desired audio type without knowing the actual class which can play the desired format. *AdapterPatternDemo*, our demo class will use*AudioPlayer* class to play various formats.

Steps

Use the following steps to implement the above mentioned design pattern.

Step 1

Create interfaces for Media Player and Advanced Media Player.

MediaPlayer.java

Step 2

Create concrete classes implementing the *Advanced Media Player* interface.

VlcPlayer.java Mp4Player.java

Step 3

Create adapter class implementing the *MediaPlayer* interface.

MediaAdapter.java

Step 4

Create concrete class implementing the *MediaPlayer* interface.

AudioPlayer.java

Step 5

Use the AudioPlayer to play different types of audio formats.

AdapterPatternDemo.java

Step 6

Verify the output.

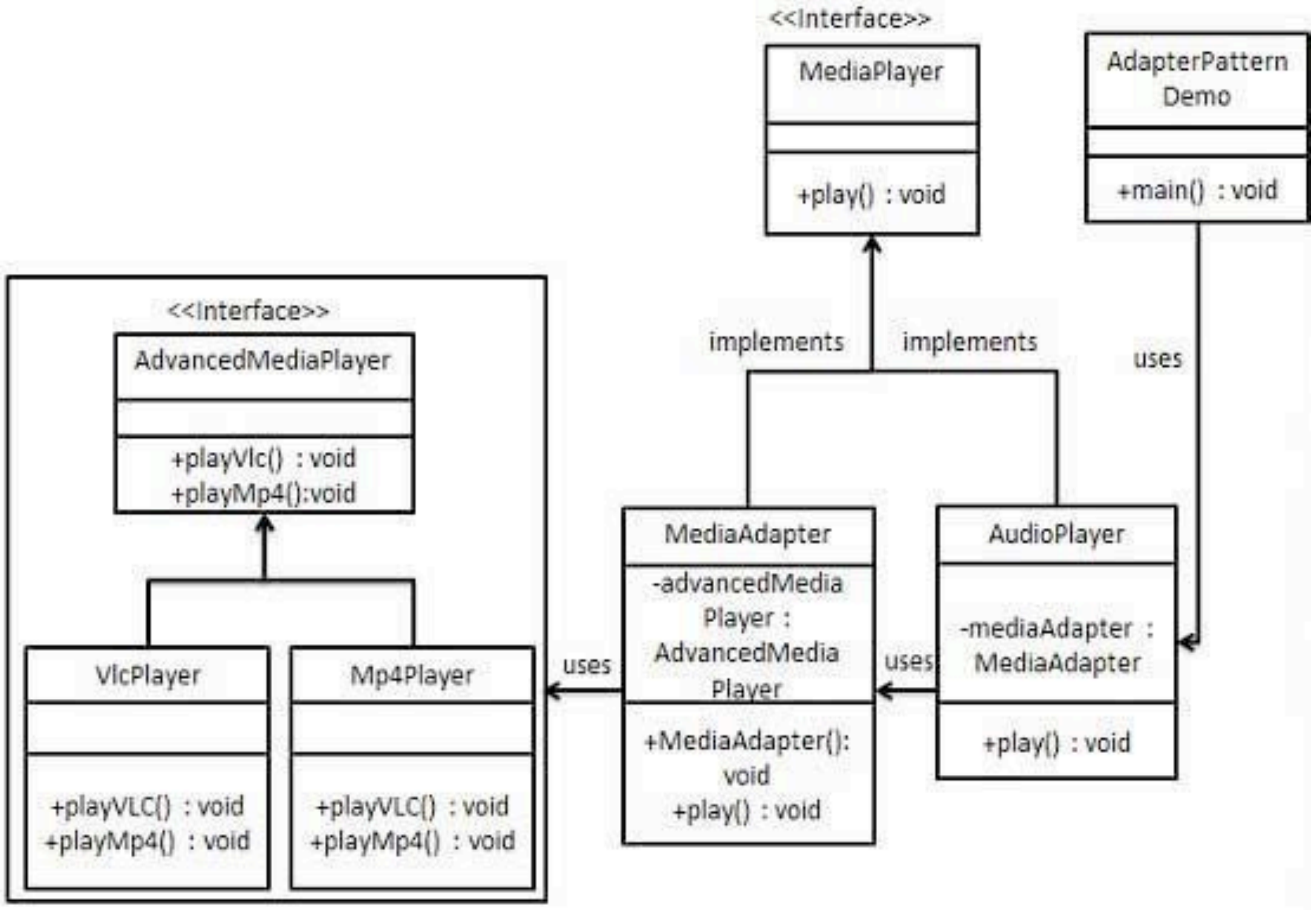
```
Playing mp3 file. Name: beyond the horizon.mp3

Playing mp4 file. Name: alone.mp4

Playing vlc file. Name: far far away.vlc

Invalid media. avi format not supported
```

Class Diagram



Evaluation and marking system:

Basic honesty in the evaluation and marking system is absolutely essential and in the process impartial nature of the evaluator is required in the examination system to become popular amongst the students. It is a wrong approach or concept to award the students by way of easy marking to get cheap popularity among the students to which they do not deserve. It is a primary responsibility of the teacher that right students who are really putting up lot of hard work with right kind of intelligence are correctly awarded.

The marking patterns should be justifiable to the students without any ambiguity and teacher should see that students are faced with unjust circumstances.

The assessment is done according to the directives of the Principal/ Vice-Principal/ Dean Academics.