Chapter - 1

Review of Object oriented Analysis and Design

"owning a hammer doesnot make one an Architect' Knowing an object-oriented language (such as Java) is a necessary but insufficient first step to create object systems. Knowing how to 'think in objects' is also a critical

The most important skill in object-oriented Analysis and Design is assigning responsibilities to objects. That determines how objects interact and what classes should perform what operation.

Analysis :

Analysis emphasizes an investigation of the problem and requirements, rather than a solution. what the problem is all about and what the system must do. "Do the right thing"

Design

Design emphasizes a conceptual solution that fulfills the requirements, rather than its implementation "Do the thing right".

OOA :

OOA is a method of analysis that examines requirements from the perspective of classes and objects found in the vocabulary of problem domain.

OOD :

It is a method of design encompassing the process of object-oriented decomposition and notation for depicting both logical and physicall as well as static and dynamic models of the system under design - Grady Booch.

Software :

It is a generic term for organized collections of collections of computer data and instructions.

↳ System Software
↳ Application Software

- System software

is responsible for controlling, itegrating, and managing the individual hardware components of a computer system.

Eg: Operating system

- Application Software

is used to accomplish specific tasks other than just running the computer system.

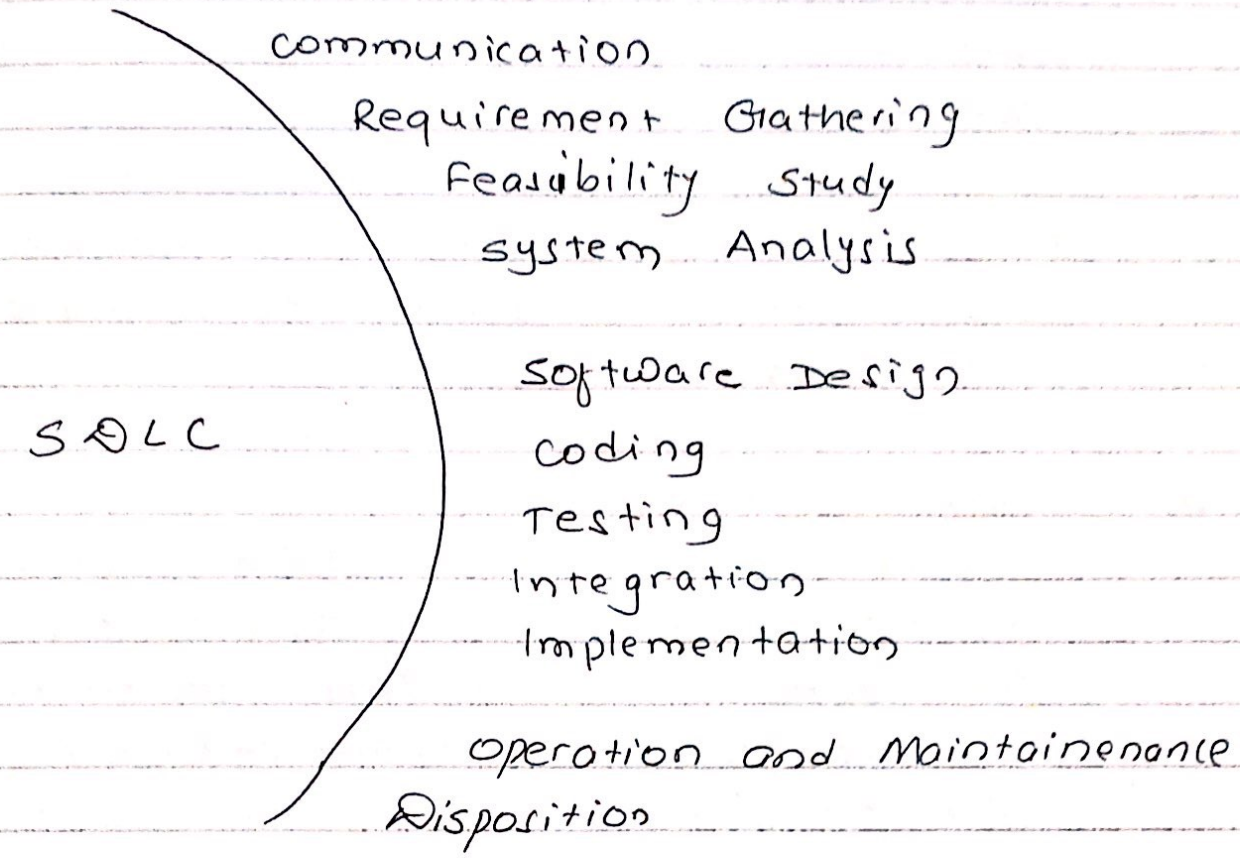
Eg: Microsoft office.

* Software Engineering

- is an engineering branch associated with development of software product using well-defined scientific principle, methods and procedures.
- is the process of solving customer's problem by the systematic development and evolution of large, high quality software systems within cost, time and other constraints.

* Software Development Life Cycle (SDLC)

- is a well defined, structured sequence of stages in software engineering to develop the intended software product.



* Software Development Paradigm

- It helps developer to select a strategy to develop the software
- A software dev. paradigm has its own set of tools, method and procedures, which are expressed clearly and defines SDLC.

They are:

1. waterfall
2. Iterative
3. Spiral
4. V model
5. Big Bang

* Patterns in Software

- Designing object oriented software is hard and designing reusable OO software is even harder.
- Experienced designers reuse solutions that have worked in the past.
- well structured OO system have recurring patterns of classes and objects.
- Knowledge of the patterns that have coorced in the past allows a designer to be more productive and the resulting designs to be more flexible and reusable.

* Design Patterns

- Design Patterns describes the relation and interactions of different class or object or types.
- They do not give the final class or types that will be used in software code directly but give an obstract view of the solution.
- Patterns show us how to build systems with good object oriented design qualities by reusing succesfull design and architectures.
- Expressing proven techniques speed up the development process and make the design pattern more accessible to developers of new system.

"The recurring aspects of designs are called Design Patterns"

A pattern is the outline of a reusable solution to a general solution to a general problem encountered in a partrular context.

- In software engineering, a design pattern is a general reusable solution to a commonly occuring problem in software design.
- A design Pattern is not a finished design that can be transformed directly into code.
- It is a description or template for 'how to solve a problem that can be used in many different situations'.
- Object oriented design patterns typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved.

A 'GOOD PATTERN' should be:
- be as general as possible
- contain a solution that has been proven to effictively solve the problem in the indicated context

Characters of design Patterns:
- They give the developer a selection of tried and tested solutions to work with.
- They are language neutral. They can be any language that supports object orientation.
- They have a proven track record as they are already widely used and thus reduce technical risk to the project
- They are highly flexible and can be used in practically any type of application or domain

Studying patterns is an effective way to learn from experience of others.

The originator of Patterns :

- Each pattern describes a problem which occurs over and over again in our env, and then describes the core of ~~sto~~ the solution to that problem, in such a way that you can use this solution a million times over without ever doing it the same way twice - Christopher Alexander 1977

Pattern Description

Each pattern is a three-part rule, which express a relation between a certain context, a problem and a solution

1. Context: The general situation in which the pattern apply

2. Problem: A short sentence raising the main difficulty.

3. Solution: The recommendation way to solve the problem in the context
   - to balance the forces'

4. Forces: The issues or concerns to consider while solving the problem

5. References: Who developed or inspired the ~~pro~~ patterns.

6. Antipatterns : Solutions that are inferior or do not work with context [optional]

7. Related patterns : Patterns that are similar to this pattern.

Importance of Design patterns :

They help us solve recurring design problems.
Note that: design patterns don't solve the problem themselves, they help us solve the problem.

- Design patters are often called time tested solutions for object oriented programing problems.

* The importance of design patterns are:
  - Identifying the right pattern for the problem will help us to avoid costly changes, un-maintainable, complex and in-efficient code as the system scales up.

→ Communication, Learning and Enhanced Insight
  - Over the last decade, design patterns have become part of every developer's vocabulary.
  - This really helps in communication. one can easily tell another developer on the team, "I've ~~been~~ used command patterns here" and another developer understands not just the design, but can also figure out the rationale behind it.
  - This helps in providing developers with better insight about parts of the application or 3rd party framework they use.

→ Decomposing system into objects
- The hard part of OO is to find objects and decomposing a system. one has to think about encapsulation, granularity, dependencies, flexibility, performance, evolution, reusability and so on.
- Design pattern really helps identify less obvious abstraction

→ Determining object Granularity.
→ Specifying object Interfaces
→ Designing for change
→ Relating run-time and compile time structure
→ Ensuring right reuse mechanism
→ Specifying object Implementation.

Classification of Design Patterns:

Design patterns were originally classified into 3 types:
- Creational Patterns.
- Structural Patterns.
- Behavioural Patterns.

extra: - Concurrency Patterns [since later been added]

A. Creational Patterns:

- creational design patterns are design patterns that deal with object creation mechanisms, trying to create objects in a manner suitable to the situation.
- The basic form of object creation could result in design problems or added complexity to the design.

- creational design patterns solve this problem by somehow controlling this object creation.

types of creational Design patterns

1. Factory Method:
   creates an instance of several derived class.
2. Singleton:
   A class of which only a single instance can exist.
3. Abstract Factory:
   Creates an instance of several families of class.
4. Prototype:
   A fully intialized instance to be copied or cloned.
5. Builder:
   separates object construction from its representation.

B. Structural Patterns:

- Structural design patterns are design patterns that ease the design by identifying a simple way to realise realationships between entities.
- These describe how objects and classes combine themselves to form a large structure.

types of structural patterns:

1. Facade:
   A single class that represents an entire subsystem
2. Decorator:
   Add responsibilities to objects dynamically.
3. Composite:
   A tree structure of simple and composite objects.
4. Adapter:
   Match interfaces of different classes.

5. Flyweight:
A fine-grained instance used for efficient sharing.

6. Proxy:
An object representing another object

7. Bridge:
separates an object's interface from its implementation.

C. Behavioural Patterns:

- Behavioural Design Patterns is a design patterns that identify common communication patterns between objects and ~~rati~~ realize these patterns
- These patterns increase flexibility in carrying out this communication

~~types of~~ types of Behavioural design patterns:

1. Interpreter:
It handles the expressions in grammars.

2. Iterator
visits member of a collection in a sequential fashion

3. Mediator
captures behaviour among peer objects without building a dependency between the objects

4. Observer
A way of notifying change to a number of clas by a single object.

5. State
Alter an object's behaviour when its state changes

6. Chain of Responsibility
A way of passing a request between a chain of objects

7. Command
   captures function flexibility
8. Template:
   captures basic algorithm, allowing variability
9. Momento:
   captures and restore the object's internal state
10. Strategy:
    Encapsulate an algorithm inside a class.
11. Visitor
    Defines a new operation to a class without change.

* Gang of Four (GOF)

In 1994, Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides published a book titled Design Patterns - Elements of Reusable object-oriented Software which initiated the concept of Design Pattern in Software Development.

Those authors are collectively known as Gang of Four (GOF). According to these authors design patterns are primarily based on the following principles of object oriented design.

* As per 'this' book, there are 23 design patterns which be classified in three categories: Creational, Structural and Behavioural patterns.

Types of Pattern

- Architectural pattern
- Design Patterns
- IDIOMS

Creational Patterns:

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

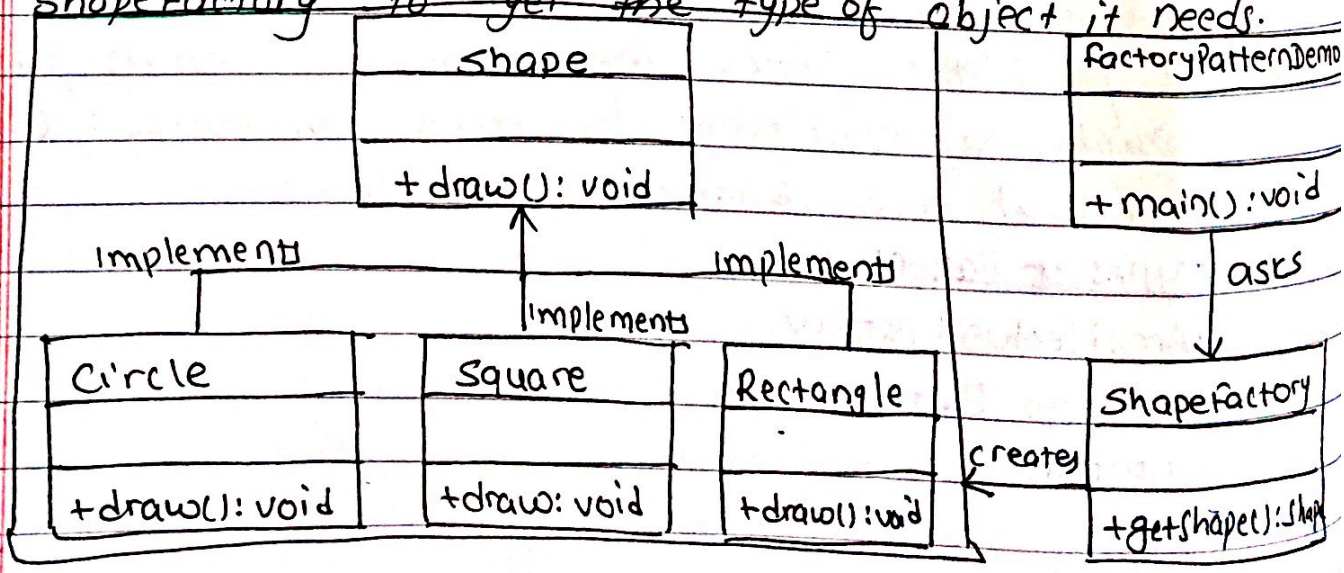
1. Factory Design Pattern

In factory pattern, we create objects without exposing the creation logic to the client and refer to newly created object using a common interface.

a. Implementation:

We are going to create a Shape interface and concrete classes implementing the Shape interfaces. A factory class ShapeFactory is defined as a next step.

FactoryPatternDemo, our main class (demo class), will use ShapeFactory to get a shape object. It will pass information (circle, Rectangle, Square) to ShapeFactory to get the type of object it needs.



step1: create an interface

Shape.java

```
public interface Shape {
    void draw();
}
```

step 2: create concrete classes implementing same interface

Rectangle.java

```
public class Rectangle implements Shape {
    @override
    public void draw() {
        sop("Inside Rectangle::draw() method.");
    }
}
```

Square.java

```
public class Square implements Shape {
    @override
    public void draw() {
        sop("Inside Square::draw() method.");
    }
}
```

Circle.java

```
public class Circle implements Shape {
    @override
    public void draw() {
        sop("Inside circle::draw() method.");
    }
}
```

Step3 : Create a factory to generate object of concrete class based on given information.

ShapeFactory.java

```
public class ShapeFactory {

   // use getShape method to get object of type shape
   public Shape getShape(String shapeType) {
      if(shapeType == null)
         return null;
      else if(shapeType.equalsIgnoreCase("CIRCLE")) {
         return new Circle();
      else if(shapeType.equalsIgnoreCase("RECTANGLE"))
         return new Rectangle();
      else if(shapeType.equalsIgnoreCase("Square"))
         return new Square();
      return null;
   }
}
```

Step4 : Use the factory to get objects of concrete class by passing an information such as type

FactoryPatternDemo.java

```
public class FactoryPatternDemo {

   public static void main(String[] args) {
      ShapeFactory sf = new ShapeFactory();

      // get an object of circle and call its draw method
      Shape shape1 = sf.getShape("CIRCLE");
      shape1.draw();
```

```
        //get an object of Rectangle and call its draw method
        Shape shape2 = sf.getShape("RECTANGLE");
        shape2.draw();

        //get an object of Square and call its draw method
        Shape shape3 = sf.getShape("SQUARE");
        shape3.draw();
    }
}
```

Steps: Verify the output

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

2. Abstract factory Patterns

Abstract factory patterns work around a super factory which creates other factories. Abstract factory is also called factory of factories.

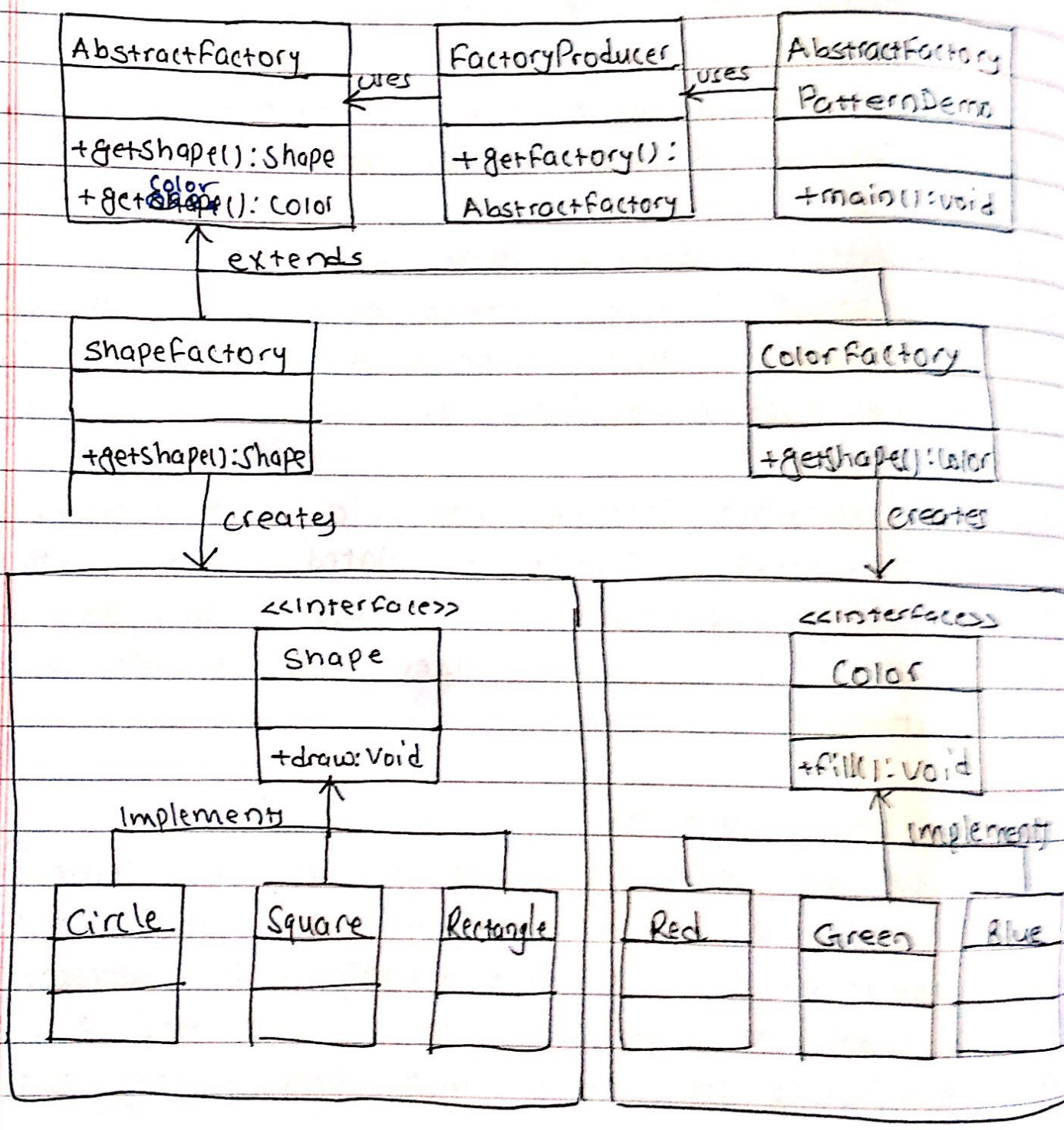
In Abstract factory pattern, an interface is responsible for creating a factory of related objects without explicitly specifying their classes. Each generated factory can give the objects as per the factory Pattern.

a. Implementation

We are going to create a shape and color interfaces and concrete classes implementing these interfaces. We create 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 an 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 AbstractfactoryFactory to get the type of object it needs.



1: Creates an interface for shapes

shape.java

```
public interface Shape {
    public void draw();
}
```

2. create concrete classes implementing the same interface

Rectangle.java

```
public class Rectangle implements Shape {
    @ public void draw {
        Sop(" Inside Rectangle :: draw().");
    }
}
```

Square.java

```
public class Square implements Shape {
    @Override
    public void draw(){ sop("Inside Square::draw().");}
}
```

Circle.java

```
Public class Circle implements Shape {
    @override
    public void draw(){ sop("Inside circle:draw().");}
}
```

3. Creates an Interface for colors

Color.java

```
public interface color {
    public void fill();
}
```

4. Create Concrete classes implementing the same Interface

Red.java

```
public class Red implements Color {
    @overrid public void fill(){ sop(" Red::fill().");}
}
```

Green.java

```
public class Green implements Color{
  @Override public void fill() { sop(" Green::fill().");}
}
```

Blue.java

```
public class Blue implements Blue {
  @override public void fill() { sop(" Blue::fill().");}
}
```

5. Create an Abstract class to get factories for color and shape objects.

AbstractFactory.java

```
public abstract class AbstractFactory {
    public abstract Shape getshape(String shapeType);
    public abstract Color getColor( String color);
}
```

6. Create factory classes extending AbstractFactory to generate object of concrete class based on given information.

ShapeFactory.java

```
public class ShapeFactory extends AbstractFactory {
    @Override
    public Shape getshape( String shapeType) {
      if( shapeType == null ) return null;
      if( st.equals("RECTANGLE")) return new Rectangle();
      else if(st.equals("CIRCLE")) return new Circle();
      else if(st.equals("SQUARE")) return new Square();
      return null;
    }
```

```
@override
public color getColor(string color){
        return null;
      }
}
```

ColorFactory.java

```
public class colorFactory extends AbstractFactory {
   @Override
    public shape getshape(string getShapeType){
       return null;
    }

    @override
    Color getColor(String color){
       if(color==null) return null
       if(color.equals("RED")) return new Red();
       else if(color.equals("GREEN")) return new Green();
       else if(color.equals("BLUE")) return new Blue();
       return null;
    }
}
```

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

FactoryProducer.java

```
Public class FactoryProducer {
  public static AbstractFactory getFactory(String Choice){
    if(Choice.equals("SHAPE")) return new ShapeFactory();
    else if(Choice.equals("COLOR")) return new ColorFactory();
    return null;
```

8. Use the FactoryProducer to get AbstractFactory in order to get factories of concrete classes by passing an information such as Type.

```
AbstractFactoryPatternDemo.java
public class AbstractFactoryPatternDemo {
   public static void main(String[] args) {

      //get shape factory
      AbstractFactory shapefactory = FactoryProducer.
            getFactory("SHAPE");
      //get an object of shape Circle, rect, square & draw
      Shape s1 = shapeFactory.getShape("CIRCLE");
      s1.draw();

      Shape s2 = shapeFactory.getShape("RECTANGLE");
      s2.draw();
      Shape s3 = shapeFactory.getShape("SQUARE");
      s3.draw();

      //get Color factory
      AbstractFactory colorfactory = FactoryProducer.
            getFactory("COLOR");
      Color c1 = colorfactory.getColor("RED");
      c1.fill();
      Color c2 = colorfactory.getColor("GREEN");
      c2.fill();
      Color c3 = colorfactory.getColor("BLUE");
      c3.fill();
   }
}
```

g. verify the output

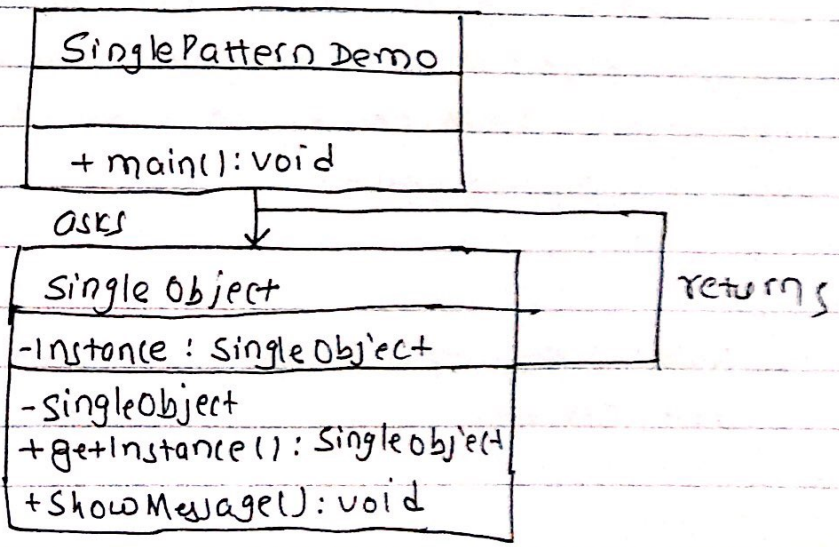
Inside circle:: draw().
Inside Rectangle:: draw().
Inside Square:: draw()
Red:: fill():
GREEN:: fill()
BLUE:: fill()

3. Singleton Pattern

Singleton Pattern involves a single class which is responsible to create an object while making sure that only single object gets created. This class provides a way to access its only object which can be accessed directly without instantiating the object of the class.

a. Implementation

- we are going to create a singleObject class which has its constructor ~~and~~ as private and a static instance itself.
- SingleObject class provides a static method to get its static instance to outside world. ~~Singleton Pattern~~ SingletonPatternDemo, our demo class, will use SingleObject class to ge a singleObject's object.



Step 1: create a singleton class

SingleObject.java

```
Public class SingleObject {
    //create an object of singleObject
    private static singleObject instance = new
        SingleObject();
    //make the constructor private so that this
     class cannot be instantiated
     private singleobject() {

     }

     // Get the only object available
     public static singleObject getInstance() {
        return instance;
     }

     public void showMessage() {
        sop(" Hello from SingleObject ");
     }
}

Public class singletonPatternDemo {
   public static void main(String args[]) {
      // get the only object available
       SingleObject object = SingleObject.getInstance();
       object.showMessage();
   }
}
```

Note: The constructor is private so we can not create object by new singleObject();

5. Builder Pattern

Design pattern

A design pattern is general repeatable solution to commonly occuring problem in software design. A design pattern is not a finished design that can be a transformed directly into code. It is a description or template for how to solve a problem that can be used in many different solutions.

Uses of design Patterns :

- It speed up the development process by providing tested, proven development paradigm.
- It improves code readability for coders
- It provides general solutions, documented in a format that doesnot require specific tied of particular solution.

① Creational patterns

In software engineering, creational design patterns are design patterns that deal with object creation mechanisms, trying to create objects in a manner suitable to the situation. The basic form of object creation could result in design issues or added complexity to the design. Creational design patterns solves this problem by somehow controlling this object creation.

- Abstract factory : creates an Instance of several families of classes
- Builder : separates object construction from its representation.
- factory method : creates an instance of several derived classes

Christopher Alexander 1970 - develop Design pattern
Gang of Four GOF 1990s

- prototype: A fully intialized instance to be copied or cloned.
- Singleton: A class of which only a single instance can exist.

Context: कहाँ situation मा Pattern use गर्ने ?
Problem: के के गर्दा design issues आएको ?
Forces: कुन चिजहरूलाई ध्यान गर्नुपर्छ, ~~कुन~~ समस्या समाधान गर्नको निम्ति।
Solution: कुन तरीकाले माथि आएको समस्या समाधान हुन्छ

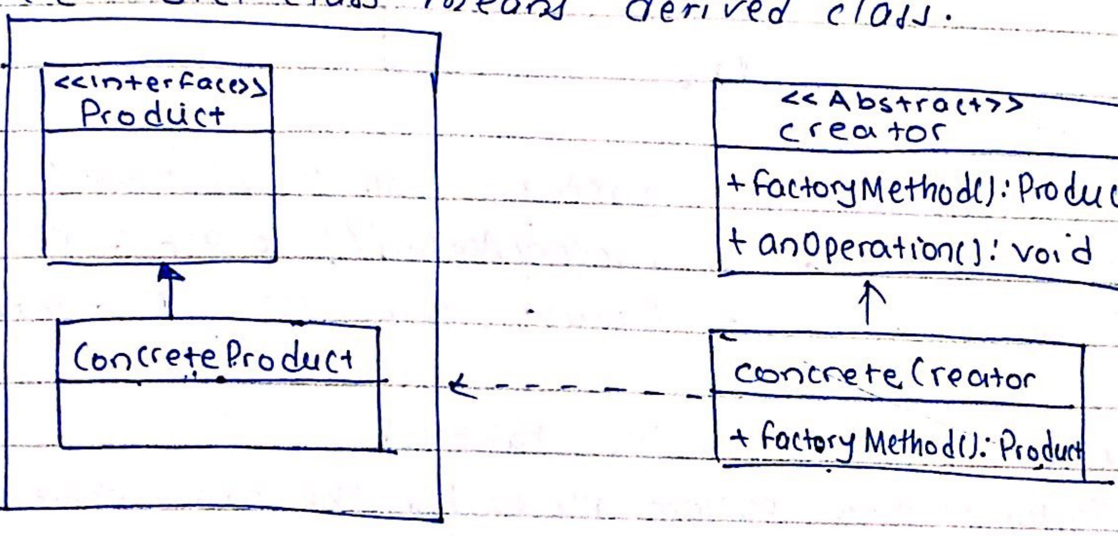
Factory Design Patterns (creating instance of derived)

- factory is just something that produces the item such as cars, TVs.
- Factory method is used to construct objects such that they can be decoupled from implementing system.
- Define an Interface for creating an object, but let the subclasses decide which class to instantiate. The factory method lets a class defer instantiation to subclasses.
- Description

a. Context: A reusable framework that needs to create object as a part of it works. However, the class of created objects will depend on application
b. Problem: How do you enable a programmer to add new application-specific class into system built on such framework
c. Forces: we want to have the frameworkclass create of application specific classes that the framework does not yet know about.
d. solution:

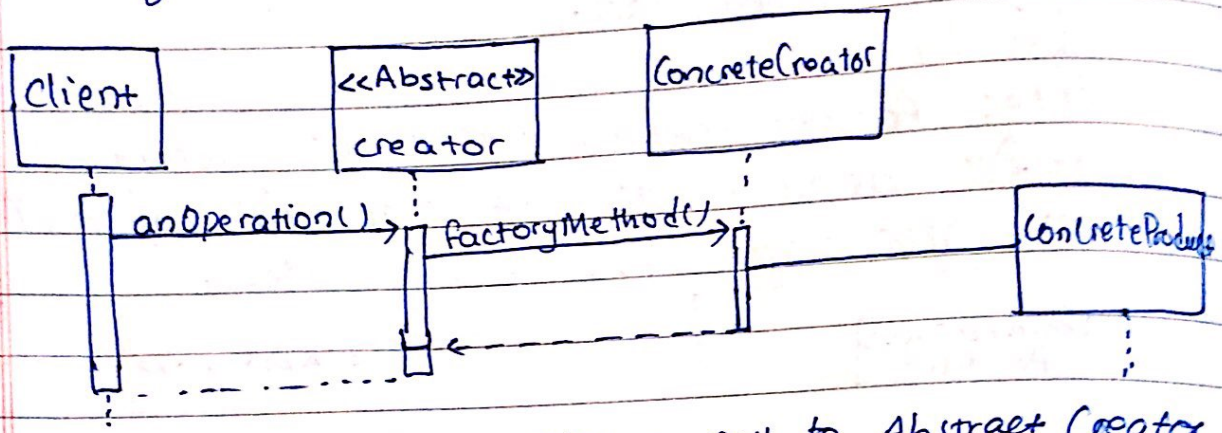
a. framework delegates the creation of instances of Application specific classes to a specialized class, the factory
b. factory is generic interfaces that defined framework.
c. factory declares a method whose purpose is to create some subclass of App specific class of ~~class~~ genericClass.

Note: factory method मा एउटा factory class बनाउने त्यसको methods ले अरु class को object दिन्छ तर अरु class means derived class.




```
public interface product { }
public class concreteproduct implement Product { }
public class concreteCreator extends creator
        protected Product factoryMethod {
                Product pr = new ConcreteProduct();
                return pr;
        }
}
public class Creator {
        main() {
            Product P1 = new ConcreteCreator().factory
                                              Method();
        }
}
```

Now lets take look at the diagram definition of the factory method. The 'creator' hides the creation and Instatiation of product from client. This is a benefit to the client as they are now insulated from any future change - the creator will look after all creation logic, allowing to decoupling.



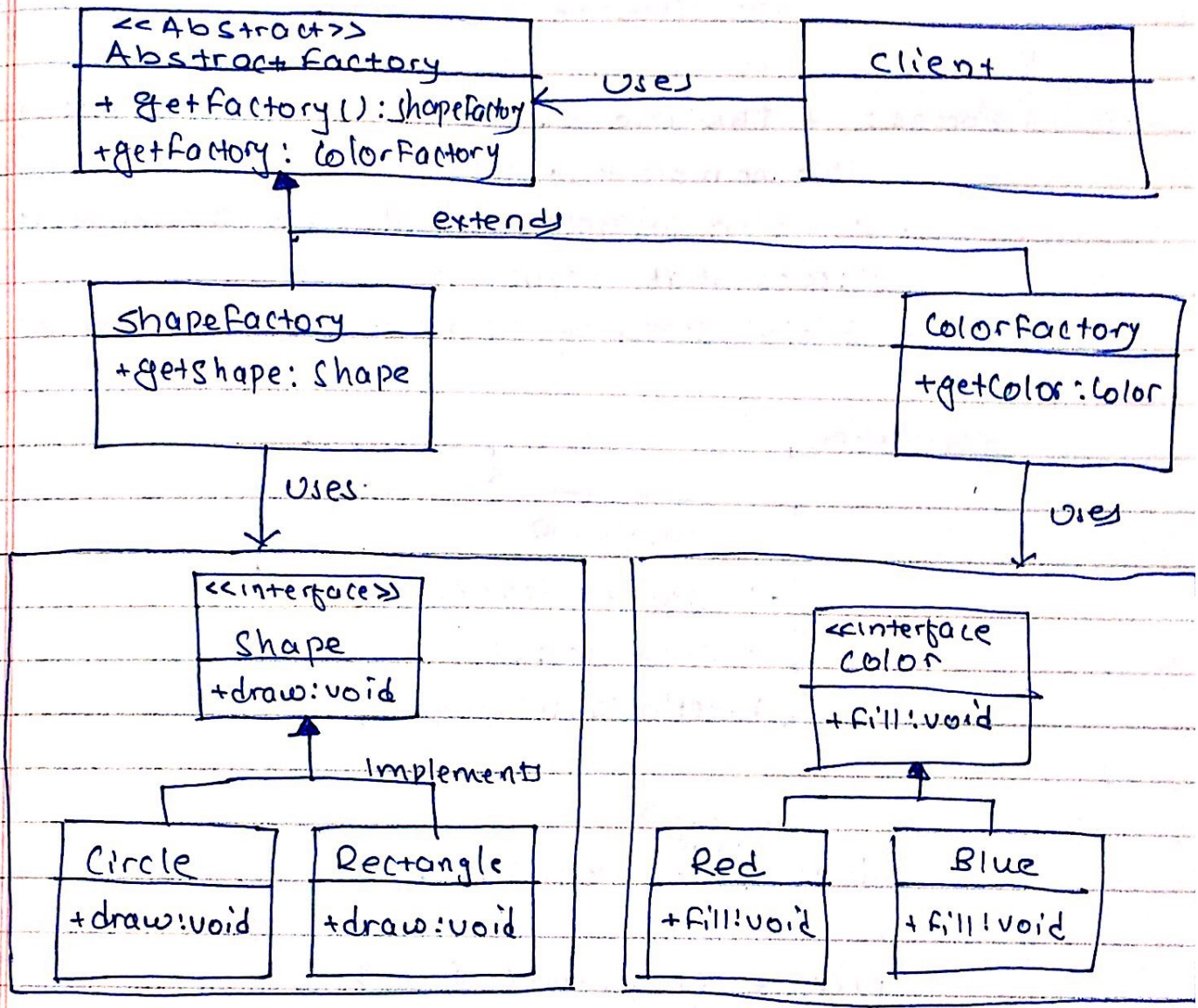
Here we see client making a call to Abstract Creator, which then uses the FactoryMethod() to get a new instance of ConcreteProduct, completes anOperation().

Where would I use this Pattern.
FactoryMethod pattern allows for the case where a client doesnot know ~~the~~ what concrete classes, it will be required to create at runtime, but just want to get a class that do the job. But lets the subclasses decide which implementation of concrete class to use.

→ Abstract Factory method are implemented with FactoryMethod can be implemented using Prototype.
→ Factory method → inheritance, Prototype : creation through delegation.

Abstract Factory Method : (creating families)

provides an interface for creating families of related or dependent objects without specifying their concrete classes.



→ The main benefits of this pattern is that the client is totally decoupled from concrete classes. Although new product family could be easily added into system

→ provide an interface for creating families of related or dependent object without specifying their concrete classes

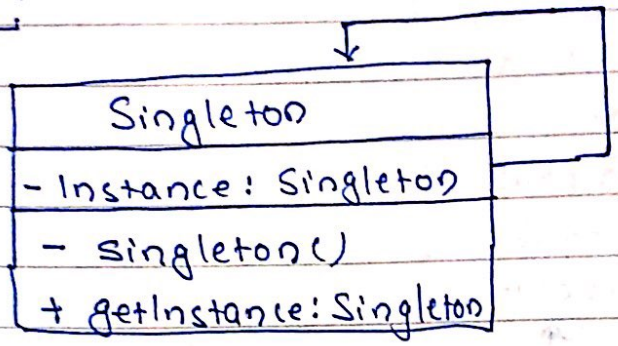
C. The Singleton Pattern : (creating one exactly...)

Context: It is very common to find classes for which only one instance should exist

Problem: How do you ensure that it is never possible to create more than one instance of a Singleton class

Forces: - The use of public constructor cannot guarantee that no more than one instance will be created
- Singleton instance must also be accessible to all classes that require it
- Global access of Singleton class is necessary

Structure:




```
class Singleton {
    private static Singleton instance;

    private Singleton() {---}

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

The Singleton pattern defines a getInstance operation which exposes the unique instance accesible by clients. getInstance() ensures that only one way of instantiating.

1. Define a private static member variable of myClass
2. make the default constructor private
3. Define a public static method to access the member

Example: In a system, there should be only one window manager to close, minimize or maximize and accessible from all the system.

- It provides a global access point to that instance
- only one instance is created.

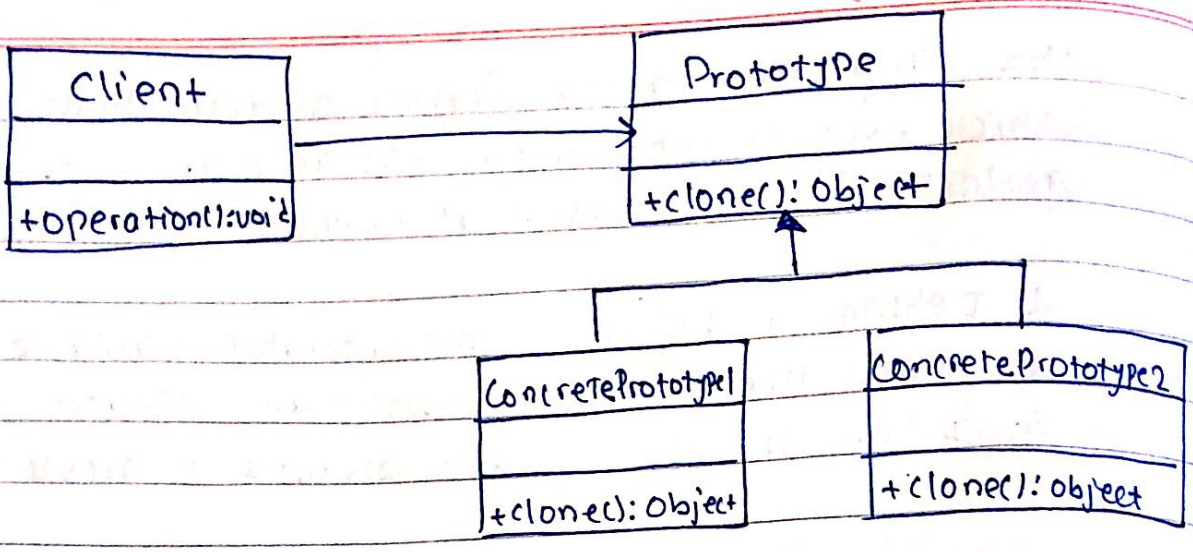
d. Prototype pattern (mix and match).

- prototype pattern involves copying something that already exist
- create objects based on a template of an existing code through cloning

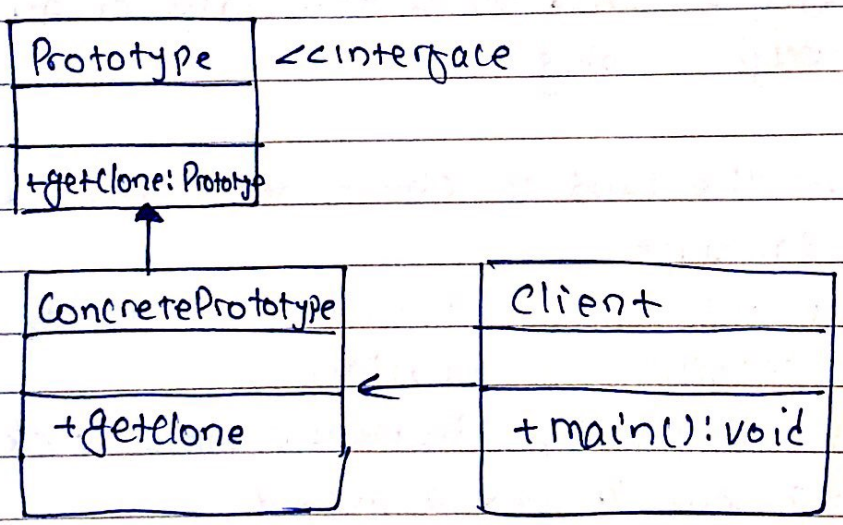
Intent:

- Specifying the kind of objects to create using a prototypical instance
- create a set of almost identical objects whose type is determined at runtime
- Assume that a prototype instance is known; clone it whenever a new instance is needed

The process of cloning starts with an initialized and instatiated class. The client ask for a new object of that type to prototype class. A ConcretePrototype, depending of the type of object is needed, will handle the cloning through clone(), making a new instance of itself.



- Prototype pattern says that "cloning of an existing object instead of creating new one and can also be customized as per the requirement.
- If the cost of creating object is expensive and resource intensive.
- Advantage
  - it reduces the need of sub-classing
  - hides complexities of creating object
  - it lets you remove/add objects at runtime.



prototype.java

```
interface prototype {
    public Prototype getclone();
}
```

ConcretePrototype.java

```
public class ConcretePrototype implements Prototype {
  @override
    public Prototype getclone() {
      return new ConcretePrototype();
    }
}
```

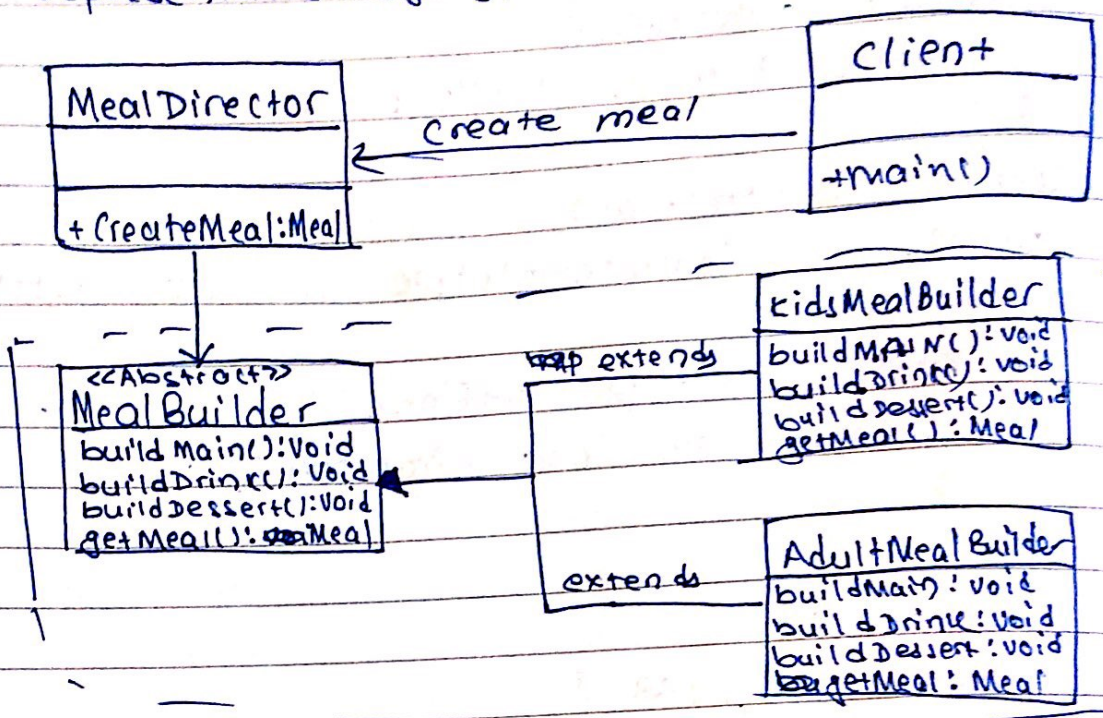
Client.java

```
class Prototypeeg {
    main() {
        ConcretePrototype CP1 = new ConcretePrototype();
        CP1.dosomething();
        ConcretePrototype CP2 = cp1.clone();
        CP2.domesty();
    }
}
```

e) Builder Design Pattern (Fast food order by kid)

- Builder pattern says that "construct a complex object from simple objects using step by step approach
- Separate the construction of complex object from its representation so that same construction process can create different representations
- It provides clear separation bet^n construction and representation of an object.
- it supports to change the
- It is used when object creation algorithm should be decoupled from the system, and multiple

representation of ~~syste~~ creation algorithm required




```
public abstract MealBuilder {
    public abstract void buildMain();
public abstract void builDrink();
public abstract void builtDessert();
public abstract Meal getMeal();
}
public class KidsMealBuilder extends MealBuilder{
}
public class AdultMealBuilder {
}
```

MealDirector

```
public class MealDirector {
    public Meal createMeal(MealBuilder builder){
        builder.builddrink();
        builder.buildMain();
        builder.buildDessert();
        return builder.getMeal();
    }
}
```

GHJV: Gomma, Helm, Johnson, Vlissides (GOF) (1991-1994)
Design Patterns: Elements of Reusable OO Software

Client.java

```
public class Client {
    main() {
        MealDirector director = new MealDirector();
        MealBuilder builder = null;
        if(isKid)
            builder = new KidsMealBuilder();
        else
            builder = new AdultMealBuilder();
        Meal meal = director.createMeal(builder);
    }
```

Structural Patterns :

- In software Engineering, structural design patterns are DP that ease in design by ~~simply~~ identifying ~~the~~ a simple way to realize relationship between entities
- Structural dp are concerned with how classes to form larger structure
- It simplifies the structure by identifying the relationship

a. Adapter pattern: Adapting an Interface into another
b. Bridge pattern: Separates an object's interface from it's implementation
c. Composite: A tree structure of simple and Composite objects.
d. Decorator: Add responsibilities to objects dynamically.
e. Facade: A single class that represents an entire subsystem
f. Flyweight: Reusing an object by sharing it
g. Proxy: An object representing another object.

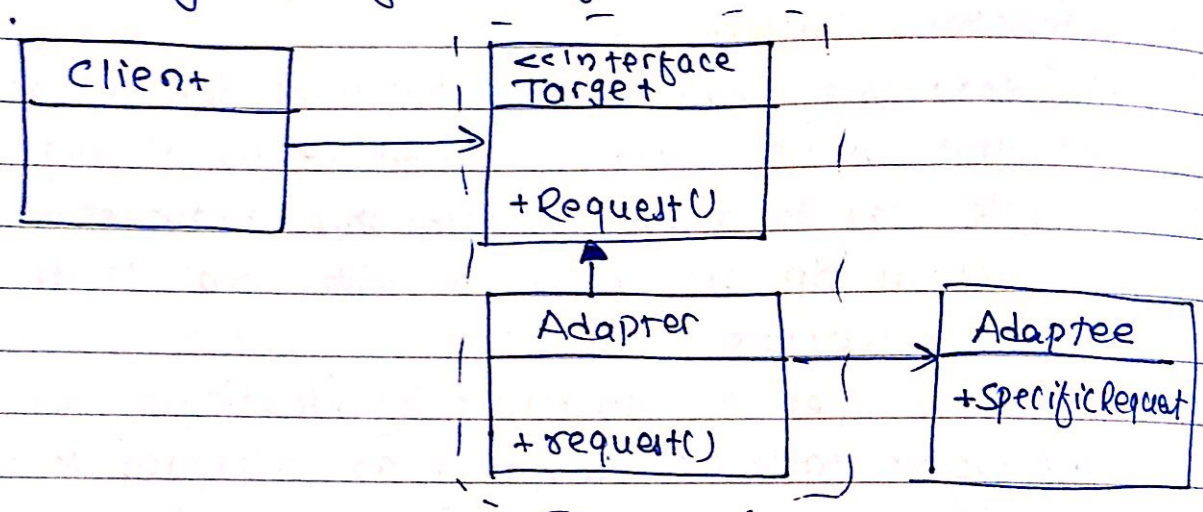
1. Adapter Pattern: [wrapper]

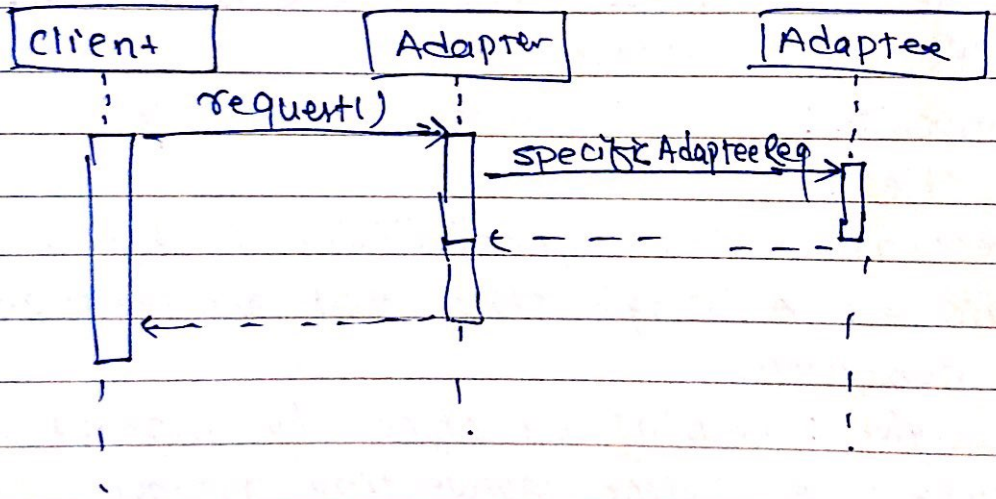
- Converts the interface of a class into another interface that clients wants.
- To provide the interface according to client requirement while using the services of a class with different interfaces.

Advantage

- It allows 2 or more incompatible objects to interact.
- It allows reusability of existing functionality.
- It is used: when an objects need to utilize an existing class with an incompatible interface

Following specification for Adapter pattern:





In background, The Adapter knows how to return a right result, when client request method to target Interface.

- The main use of this pattern is when class that you need to use doesnot meet the requirements of an Interface

Target: defines the domain specific interface that client uses

Adapter: adapts an Interface to the Target Interface

Adaptee: defines an ~~ado~~ existing interface that needs adapting.

Ex

2) Bridge Pattern [Handle or Body]

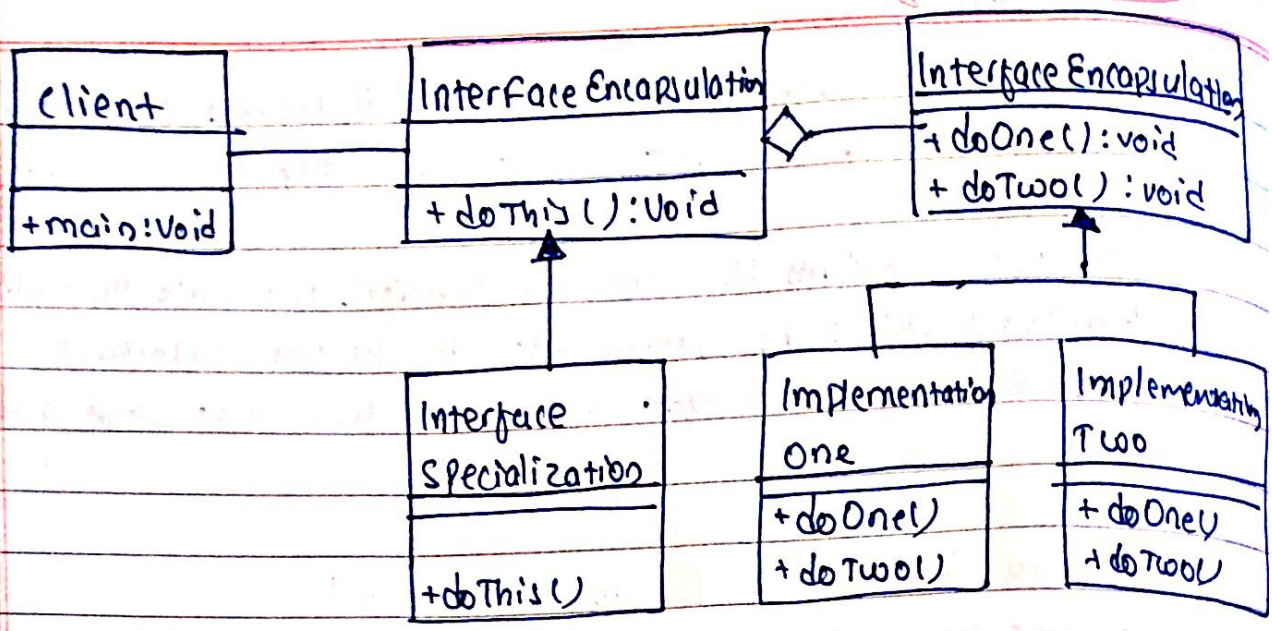
- Decouple the functional abstraction from the Implementation so that the two can vary Independently.

- Advantage

- It enables the separation of Implementation from Interface.
- It improves the extensibility
- It allows the hiding complexities from client
- Understand: displaying different Image formats on different OS. You might have different image abstraction for both jpeg and png images. The image structure is same across all OS, but it is viewed (implemented) is different on each OS.
- Decouple an abstraction from its implementation so that the two can vary indepentdtly.

Use:

- desire of run time binding
- hiding details



In doThis { Implementation.doOne();
Implementation.doTwo();
}

Example: Bridge Pattern decouples an abstraction from its Implementation, So that the two can vary independently. A household switch controlling, lights, ceiling fans etc are examples of ~~Bridge~~ Bridge. The purpose of switch is to turn on/off.

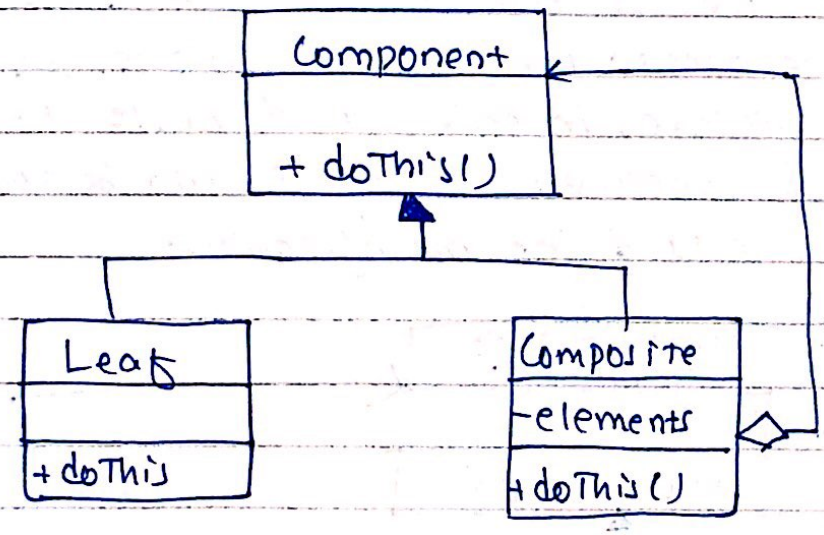
यो design Pattern मा: यहुदा class लाई अको class ले implement गर्छ, अनि आफनो way मा modify गर्ने काम गर्छ, जस्तै Switch मा on/off हुन्छ, तर अब Switch-Implementation ले switch लाई जे गरे तापनि on/off गर्न बनायो।

एउटा class ले sub class consist गर्दै 2 subclass वा simple class हुन्छ or subclass को पनि अर्को subclass हुन्छ

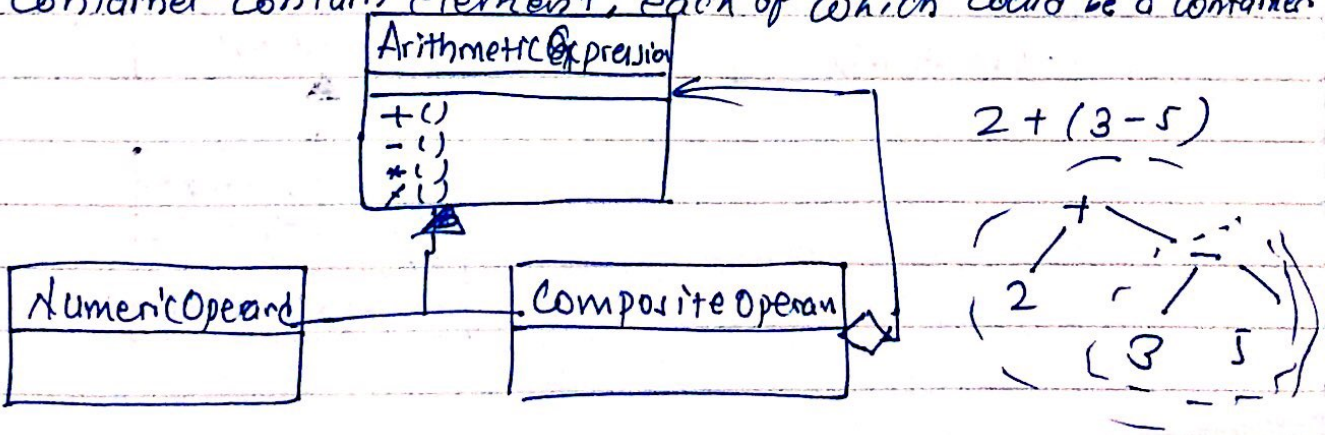
c. Composite Design Pattern

- Composite dp compose objects into tree structures to represent whole part hierarchies. Composite lets client treat individual objects and compositions of objects uniformly.
- Example : In hotel, In MenuBar has menu has many menu items, which themselves can have submenus. (organization chart, Employee hierarchy)

- Composite as a collection of objects, where any one of these objects could itself be a Composite or a simple objects.
- recursive composition
- '1 to many' 'has-a' up the 'is-a' hierarchy



- Directories that contain files, each of which could be directory.
- Container contain element, each of which could be a container



Describe each :-
The Intent of this pattern is to Compose objects in tree structures to represent part-whole hierarchy

Decorator Pattern →

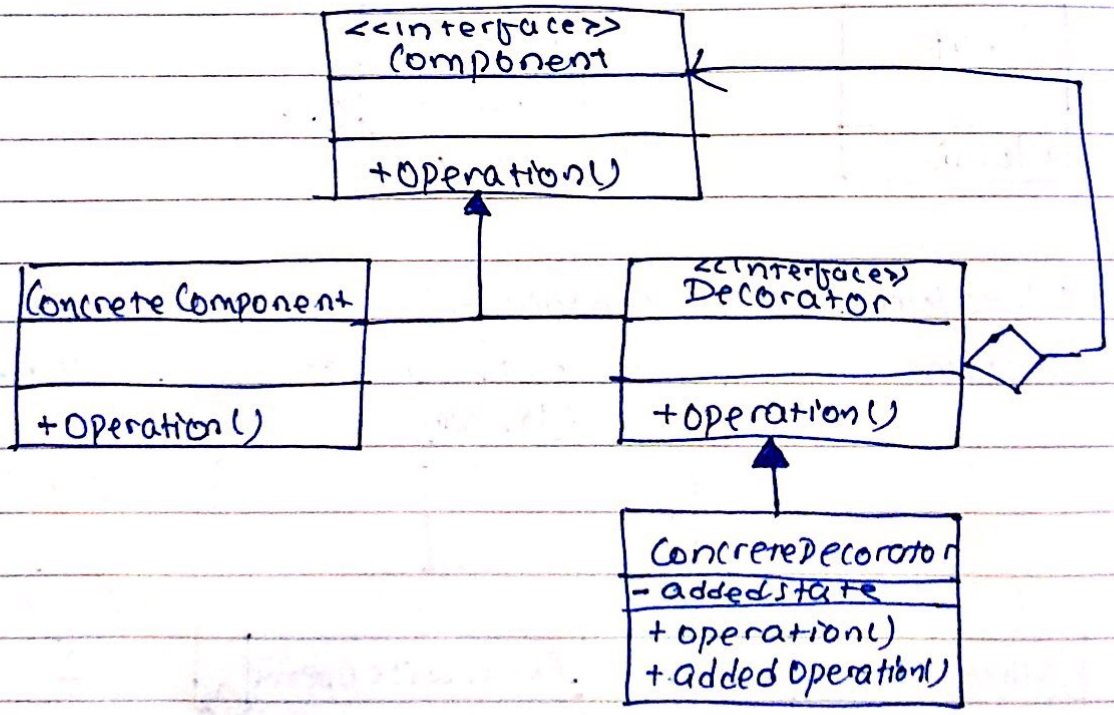
- It allows class Behaviour to be extended dynamically
- Attach additional responsibilities to an object dynamically
- Decorator provide a flexible alternative to subclassing for extending functionality.

Problem : You want to add behaviour to individual objects. Though Inheritance ~~added~~ applies to entire class and is static

The Concept of decorator is that it adds additional attributes of responsibilities to an object dynamically.

Ex: The picture is our object which has its own characteristics. for display purpose, we add a frame to picture, In order to decorate it.

- Open/closed principle : classes should be open for extension, closed for modification.

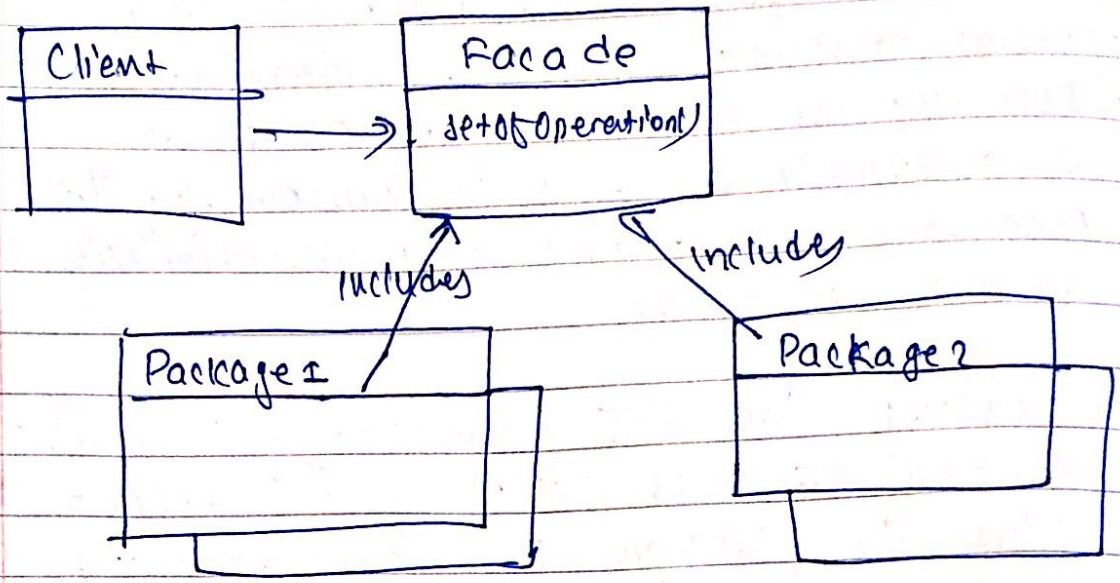


- The Component defines the interface for objects that can have responsibilities added dynamically.
- ConcreteComponent is simple implementation of this Interface
- Decorator has reference to Component, as Decorator is essentially wrapping the Component.
- ConcreteComponent just adds responsibilities to original Component.

§ Existing class को runtime में कोई method add करें। जैसे picture एक class है, runtime में picture show हो रही है, picture में frame Add हो (additional property / behaviour add)
}
It can be done by subclassing

8. Facade Pattern
- It says "just provide a unified and simplified interface to a set of interfaces in a subsystem, therefore it hides the complexities of the subsystem from the client
- GOF: Provides a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
- Insulating client from Subsystem
- Like in factory Adapter pattern, the facade can be used to hide the inner working of class. All client needs to interact with Facade, not the Subsystem

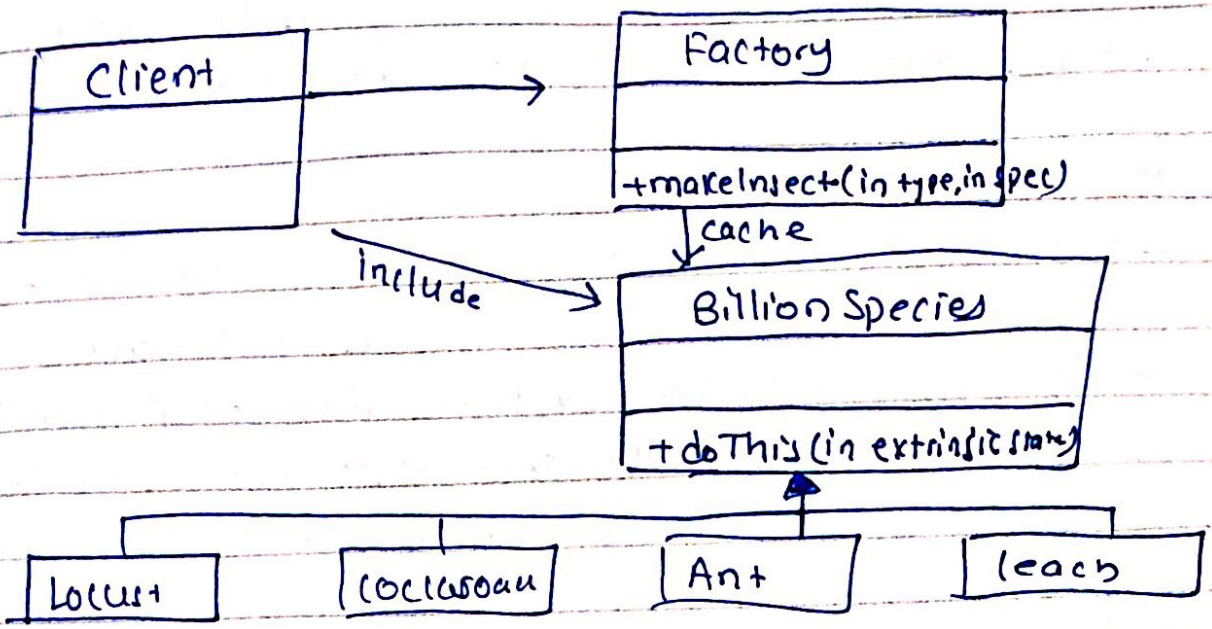
- All Abstract Factory Method are Facade.



- It Simplify the interface
- Service Oriented architecture make use of facade
- All other Packages are hidden from client
- A segment of the client community needs a simplified interface to the overall functionality of complex subsystem
- provide an interface to a package of classes
- modularizes designs by hiding complexity
- Webservices provided by a web site

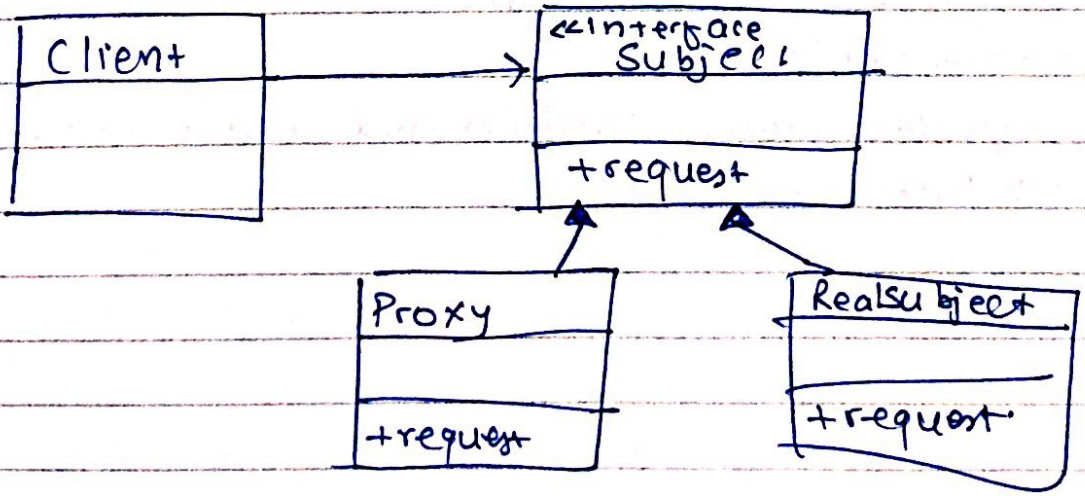
6 Flyweight: Reusing the objects by Sharing it
- Designing objects down to the lowest levels of system 'granularity' provides optimal flexibility
- The flyweight pattern describes how to share objects to allow their use at fine granularity without prohibitive cost.
- Many similar objects are used and storage cost is high
- Create a factory that can cache and reuse existing class instances
- It implements Composite to implement shared Leaf nodes

- It shields the client from complexities
- It promotes loose coupling betn subsystem and its clients
- It is used when an application uses no. of Object



Example: Modern Browser, it loads all image and save it to cache, when user scroll, it just display required image from cache.

g. Proxy :- an obj repr another obj [surrogate]
- provides the control for accessing the original object
- provides the protection to the original obj. from outside world
- In real world, Cheque or credit card is proxy for which our account ie credit card represent cash. provide means of accessing cache



Patterns scope:

- Object patterns: specify relationship between objects
  - purpose of dp is to allow instance of diff classes to be used in a same place in a pattern
- class patterns: relationship bet^n classes & their subclasses

The process of Design

- Design is a problem solving proces whose objective is to find and discribe way
- To implement the System's functional requirement
- while respecting the constraints imposed by quality, platform and proces requirement

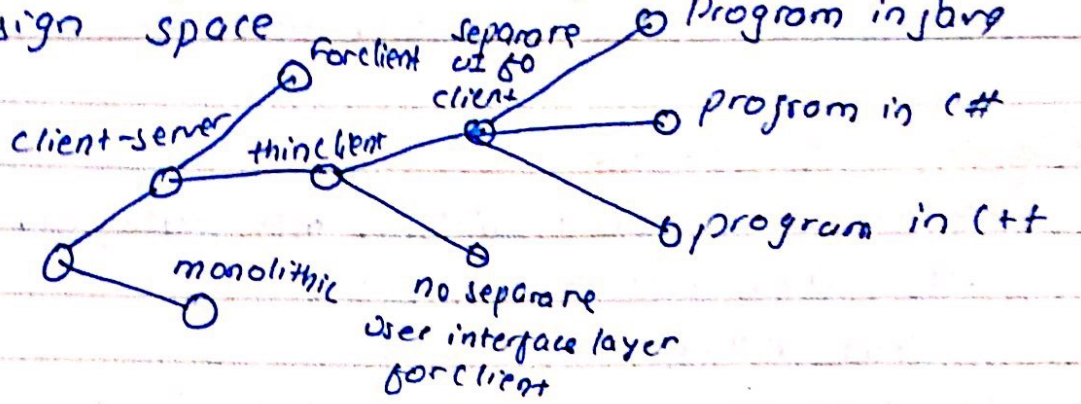
* A designer is faced with a series of design issues
  - These are sub problems of the overall design problem.
  - Each issues has several alternative solution
  - Designer makes a design decision to resolve each issues.
    - This process involves ~~the best~~ choosing the best option from among the alternatives

Design Decision

The Software Engineer uses knowledge of
a. requirements
b. design as created so far
c. technology Available
d. Software design principles and best practise
e. Past knowledge

Design Space

The space of possible designs that could be achieved by choosing different set of alternatives is ofstem called design space



Component

Any piece of software/hardware that has a clear role

- A component can be isolated, allowing you to replace it with a different component that has equivalent functionality.
- Many components are design for reusable
- Special purpose functions (Goal meet)

Module

A component that is defined at the programing Language Level.

Ex: methods, classes and packages are models in java

System

- A logical entity, having set of definable responsibilities or objective, consisting of hardware or both
- A system can have a specification which is then implemented by a collection of component
- A system continued to exist, even if its component changes.
- The goal of requirement Analysis is to determine the responsibilities of a system

Subsystem : A system that is part of larger system and and which has a definite Interface

Top down design: to provide the system, a good structure
- first do high level construct and then low level
Bottom up design: useful so that reusable component can be created. low level and then high level.

Different Aspect of Design

A. Architecture Design : division into subsystem & compone
  · How these will be connected ?
  · How will they interact ?
B. Class ~~Diagram~~ Design : various features of class
C. User Interface Design
D. Algorithm Design : computational mechanism
E. Protocol Design : communication

Design pattern Vs Architecture :

Design patterns
- Design is tactical
- well known patterns for solving technical problem in a proven way
- How classes, methods collaborate with each other
- pattern describes a problem which occurs over and over again

Architecture
- It mostly addres problem of functionality, system partitioning, protocol, security.
- Architecture is strategic
- It compromises the frameworks, tool, programs
(1) Component rather than classes

Principles leading to Good design:

Overall goals of good design are
- reducing cost and increasing revenue
- ensure that requirement are conform.
- Accelerating development.
- Usability, Efficiency, reliability, Maintainability, Reusability.

1. Divide an Conquer
2. Increase cohesion where possible
3. Reduce coupling where possible
4. Keep the level of abstraction as high as possible
5. Increase Reusability where possible
6. Reuse existing designs
7. Design for flexibility
8. Anticipate obsolescence
9. Design for portability
10. Design for Testability
11. Design defensively

Techniques for making good design decision

1. List and describe the alternatives for the design decision.
2. List advantage/disadvantage of each wrt our objective.
3. determine if any alternative prevent us from meeting one or more objective
4. Choose the alternative that helps us to best meet our objective
5. Adjust priorities

Evaluation of alternatives
- Security
- Maintainability
- cpu efficiency
- Nlw bandwidth efficiency
- memory.

Software Architecture

- Software architecture is process of designing the global organization of a software system, including
  - Dividing software into subsystem
  - Deciding how these will interact.
  - Determining their interfaces.
- The architecture is the core of the design, so all
- The architecture will often constrain the overall efficien reusability and maintainability of the system
- Software Architecture is the process of defining a structured solution that meets all of the technical and operational requirement

* Importance of Software Architecture

- better understand of the system
- allows people to work on individual pieces of the system in isolation
- To prepare for extension of system.
- To facilitate reuse and reusability
- It gives the right technical solutions to ensure your ~~access~~ success.
- If base of soft is solid, -ve impacts on time to time market, costs and adaptability.
- It is Important because :

* ~~What~~ :

a. A basis for Communication
b. Earliest Decision
c. Transferability of the model

→

When we talk about the architecture of a software, we talk about the plans that describes a set of aspects and decision that are important to a software.
- This implies taking into consideration all kinds of requirement (performance security), the org. of system, how the system parts communicate with each other, whether they are external dependencies what the guidelines and implementation technologies are, what risks to take into consideration

- Softwar Arch. should be strong and easy to maintain when we find bugs.
- domain concepts that nearly all members will understand
- flexible, extensible, scalable, usable on long term
- Refactoring should be easy
- when adding features, performance shouldn't decrease.

- People believe that, by looking a final product, one can't say it have a good Software Architecture.
- Still, we can find signs of good one
- To be a good Software Architecture, Software should be:
  a. Software is user-friendly
  b. Solution is flexible, user adapt it quite easily
  c. If soft is scalable
  d. easy to test, Performance is fast.
  e. Strong & reliable

Good Architectural Model :

- logical breakdown into subsystem
- Interfaces among the subsystem
- data will be shared among subsystem
- Components exists at run time,

* Design Stable Architecture

To ensure maintainability and readality of system, architectural model must be designed to be stable.

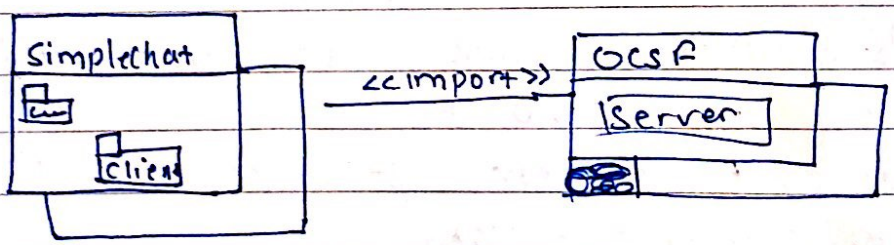
- Being stable means that the new features can be easily added with only small changes to the architecture

* Developing an architectural model

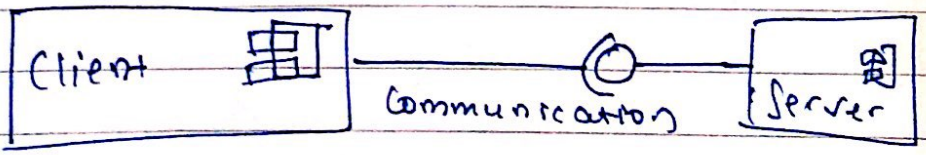
1. Start by sketching an outline of arch.
   - based on principle requirement and usecases
   - determine main component
   - choose among various architectural
   - Refine the architecture
   - mature the architecture

* Describing an architecture using UML:

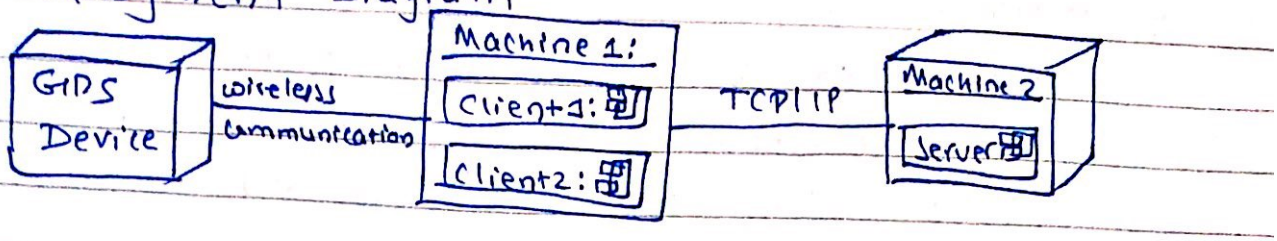
- Package Diagram



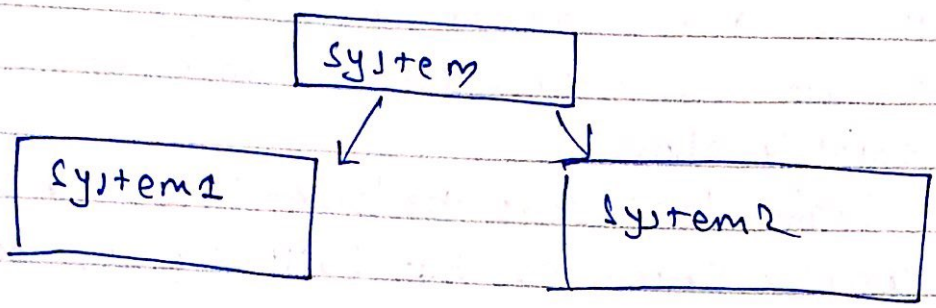
- Component Diagram



- Deployment Diagram



- Subsystem Diagram



Architectural Patterns :

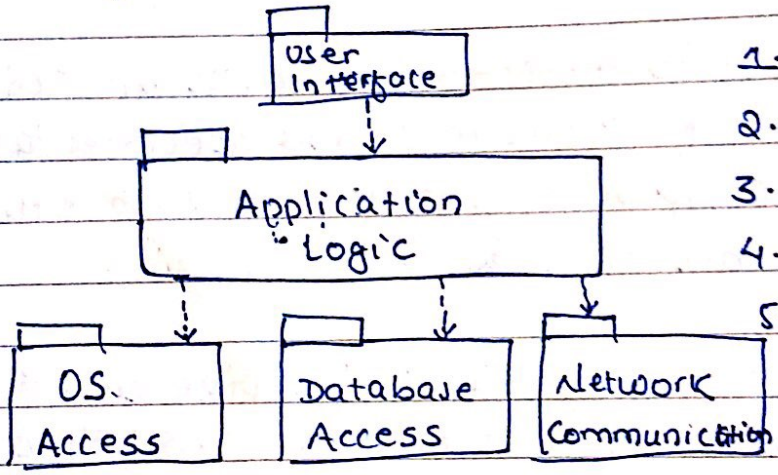
- The notion of patterns can be applied to software architecture.
- These are called architectural patterns or styles.
- Each allows you to design flexible systems using Component
- It tell us how to ~~architectur~~ organise our Code.
- highest level of granularity and specifies layer.
- how these component interact with each other
- Component are as independent as possible

Example:
Client Server, multilayered, MVC, pipe and filter, message oriented, Service oriented, Monolithic, Event Driven.

Architectural pattern is general, reusable solution to a commonly occuring problem in Soft Arch within given Context.

A. MultiLayer Architectoral Pattern :

- MLA is an arch. model that propose the org. of Software components into different layers. Each of those layers is implemented as a physically seperated containers of Software components
- In a Layered system, each layer communicates only with the layer immediately below it.
- Each layer has a well-defined interface used by layer immediately above.
- The higher level sees the lower layer as a set of Servic
- A complex System can be built by superposing layers at increasing level of abstraction.
- It is important to have a separate for UI
- Layers immediately below the UI Layer provide the application functions determined by the use case.
- Bottom layers provide general ~~accss~~ services
  Eg: Nlw communication, database acces



a) Typical layers in an Application program

1. Divide and Conquer
2. Increase Cohesion
3. Reduce Coupling
4. ~~Abs~~Increase Abstractio
5. Increase Reusability
6. Reuse exist design
7. ~~Increase~~ Anticipate chang
8. Increase Flexibility
9. ~~Re~~design for Portabil
10. design for Testabili
11. design defensive

2016 Spring

1a) Client-Server Architecture. How is it more Secure than P2P?

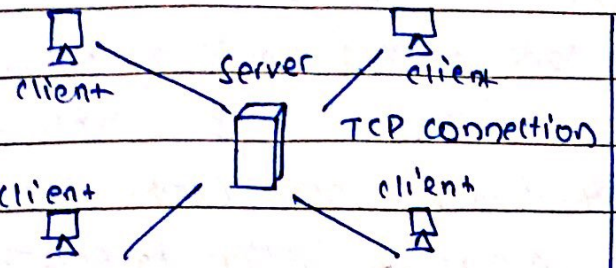


fig: client-server

Client server is a program relationship in which one program (client) requests a service or resources from another program (the server).

- The client-server model is a distributed application structure that partitions tasks or workloads between the providers of a resource or service, called servers and service requesters, called client.
- clients and servers exchange messages in a request-response pattern. The client sends a request, server responses.
- Clients /server example are: web browser, mail
- The client server architecture decreased traffic network by providing a query/response rather than a total file transfer.
- Remote procedure call (RPCs) or standard query language (SQL) are used to communicate between client and server.
- The performance is depends on processor speeds, memory, bandwidth, capacity, disk speeds, input-output devices
- The ~~eti~~ system is scalable, client can be added.
- The environment is Heterogenous and multivendor ie the os, hlw is not same betn client and server.
- client and server process communicate through API and RPCs.
- Another type of nlw architecture is peer to peer (P2P), in which computer is both client and server.

- Client Server is more secured than peer to peer NW because client server can have passwords to own individual profiles so that nobody can access anything what they want
- All the data is stored onto the servers which generally have far greater security than most client. Server can control the access and resources better to guarantee that only those clients with appropriate permissions may access and change data.
- P2P is less secure because security is handled by individual computers, not on the network as a whole.
- P2P doesnot have central storage or authentication of client, All clients are conversly dedicated servers

Client Server	Peer to Peer
1. Specific server and specific clients.	1. each node act as client or server
2. client request for server responses.	2. each node can request for service and provide services
3. Focus on sharing information.	3. Focus on connectivity.
4. The data is stored in a centralized server.	4. Each peer has its own data.
5. When too many client request, server can be bottlenecked	6. As service are provided by several servers, not bottlenecked.
6. Expensive	6. Cheap
7. Secured	7. less secured
8. stable / scalable	8. unstable /

Note:- ⊛ UML is a standard langooge for specifying, visualizing, constructing and documenting the artifacts of the software systems.

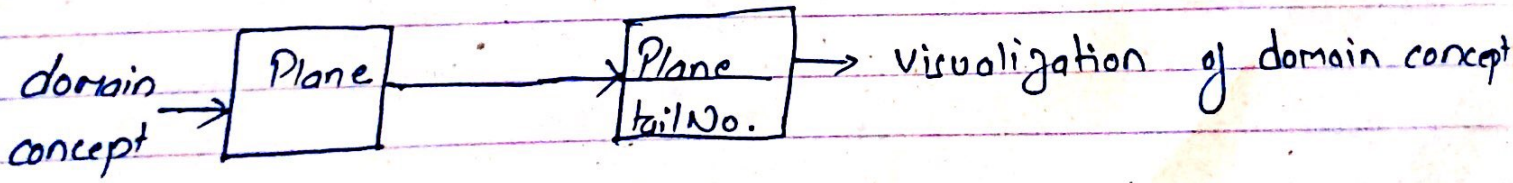
Object Oriented Design and Modelling

Q. What is object oriented analysis and design? How does object orientation emphasizes on representation of object. Describe with example.

⇒ During object-oriented analysis there is an emphasis on finding and describing the objects or concepts in the problem domain. for example, in the case of the flight information system, some of the concepts include plane, flight and pilot.

During object-oriented design there is an emphasis on defining software objects and how they collabrate to fulfill the requirements. for example a Plane software object may have a tailNumber attribute and a getFlightHistory Method.

finally during implementation design objects are implemented such as a Plane class in Java.



representation in an object oriented programming language →

```
public class Plane
{
private string tailNumber;
public list getflightHistory();
}
```

Fig:- Object Oriented emphasizes representation object.

* Analysis emphasizes in an investigation of the problem and requirements, rather than a solution. For example: if a new online trading system is desired, Analysis answers the following questions:

How will it be used?
What are its functions?

Analysis is a broad term and it is reffered to as object oriented analysis.

* Design emphasizes a conceptual solution (in software and hardware) that fullfills the requirement rather than it's implementation. Design idea often exclude low-level or obvious details obvious to intented customers. Ultimately designs can be implemented and the implementation (such as code) express the true and complete design.

They have been summarized in the phrases do the right thing (Analysis) do the things right (design).

Q. What is Unified Process (UP)?

A software development process describes an approach to building, deploying and possibly maintaining software. The UP has emerged as a popular iterative software development process for building object-oriented systems. It has 4 phases:

1. Inception: approximate vision, business case, scope, vague estimates

2. Elaboration: refined vision, iterative implementation of the core architecture, resolution of high risks, identification of most requirements and scope, more realistic estimates

3. Construction: iterative implementation of the remaining lower risk and easier elements and preparation for deployment.

4. Transition:

It has 3 disciplines:

① Business Modelling: single app ⇒ domain object modelling large scale business, dynamic modelling across entire enterprise.

② Requirements: writing use case and identifying non-functional requirements.

③ Design: All aspects of design, including the overall architecture, objects database, networking etc.

- The UP is an iterative process. Iterative development influences how to Introduce OOAID, and to understand how it is best practised.
- UP practices provide an example structure for how to do and thus how to explain OOAID.
- UP is flexible, and can be applied in lightweight and agile approach that includes practices from other agile methods.

Workflows	inception phase	elaboration Phase	construction Phase	transition Phase
Business Modelling				
Requirement Analysis				
Design				
Implementation				

i) Sequence diagram:

The sequence diagram captures the time sequence of message flow from one object to another. It shows how objects communicates with each other over time. That is sequence diagrams are used to model object interactions arranged in time sequence and to distribute use case behaviour to classes. They can also be used to illustrate all the paths a particular use case can ultimately produce. It can be illustrated as:

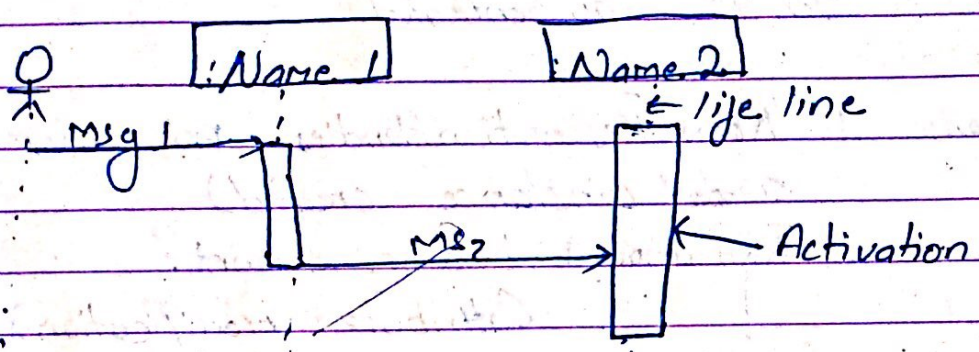


fig:- Sequence Diagram.

ii) Collaboration diagram:

Collaboration diagrams illustrate object interactions in a graph or network format, in which objects can be placed anywhere on the diagram. Collaboration diagrams show how objects interact and their roles. They represent the structural organization of object. They are best suited to the portrayal of simple interactions

among relatively small numbers of objects. It can be illustrated as:

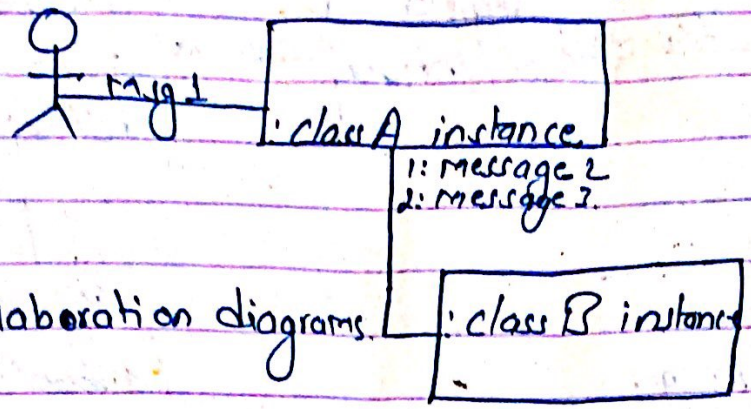


fig:- Collaboration diagrams.

They are similar as they are called interaction diagrams. Both of them display messages between objects. Both sequence and collaboration diagrams show synchronous messages.

×××

Note:- (#)

Object Oriented Design	Object Oriented Analysis
→ Real Use case.	→ Essential Use case.
→ Design class Diagram (DCD)	→ Domain modelling
→ Interaction Diagram	→ System Sequence Diagram (SSD)
→ Sequence Diagram	
→ Collaboration Diagram	

for Object Oriented Analysis:

(a)

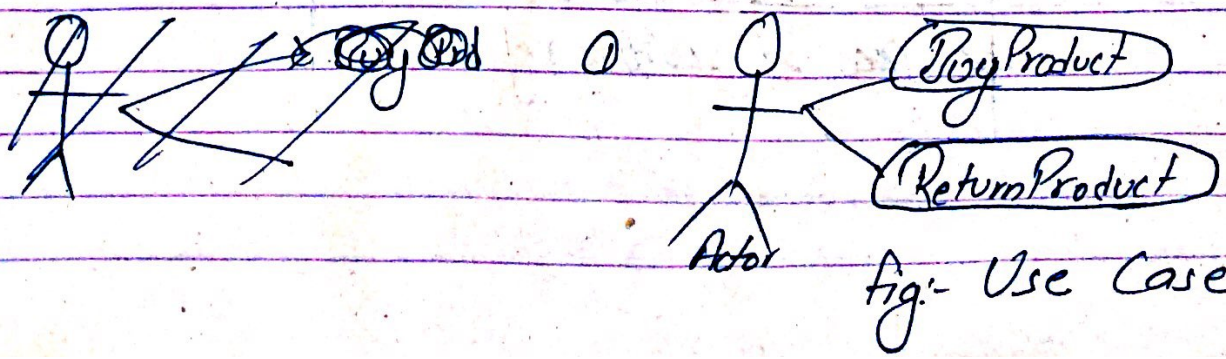


fig:- Use Case.

(ii)

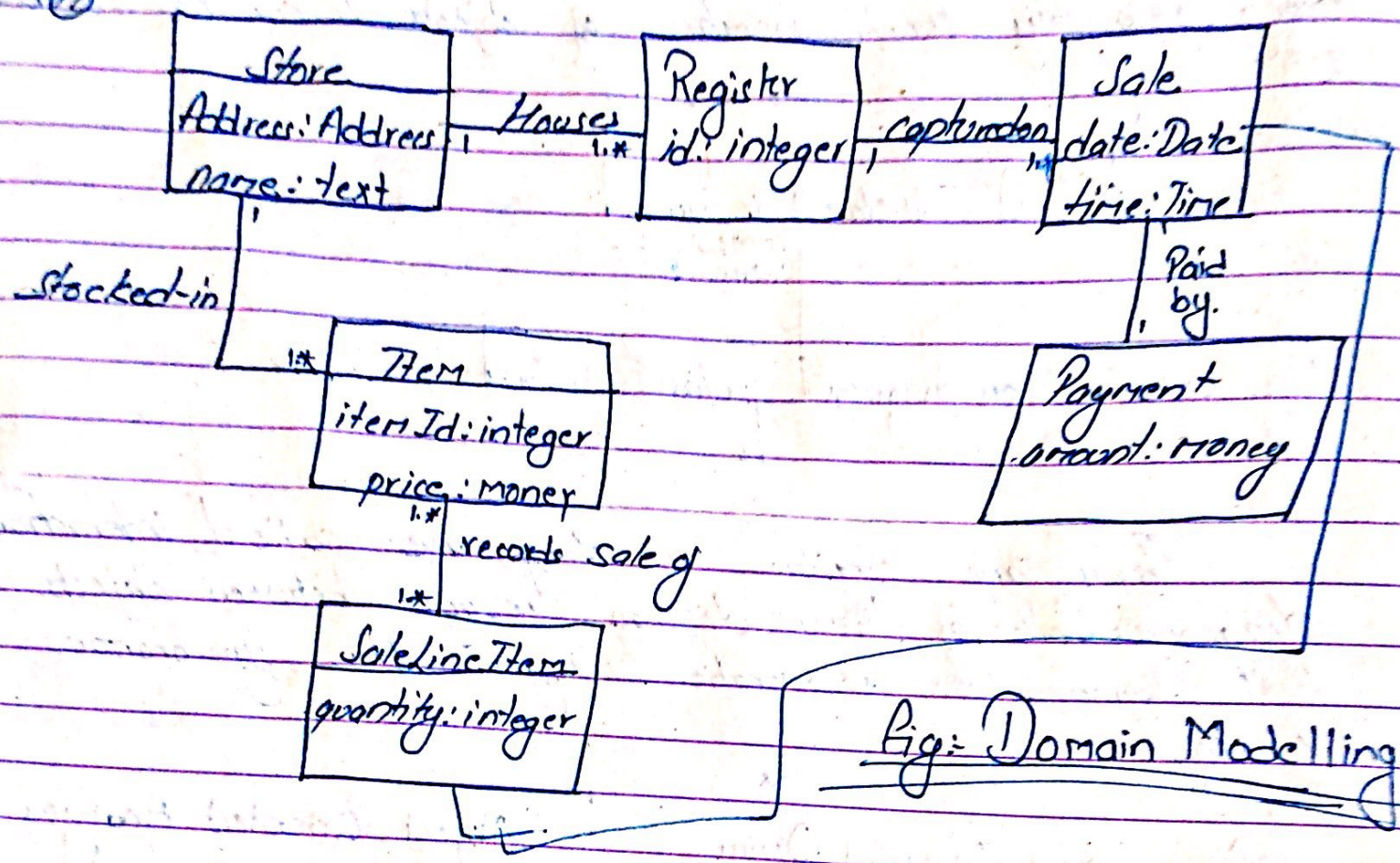
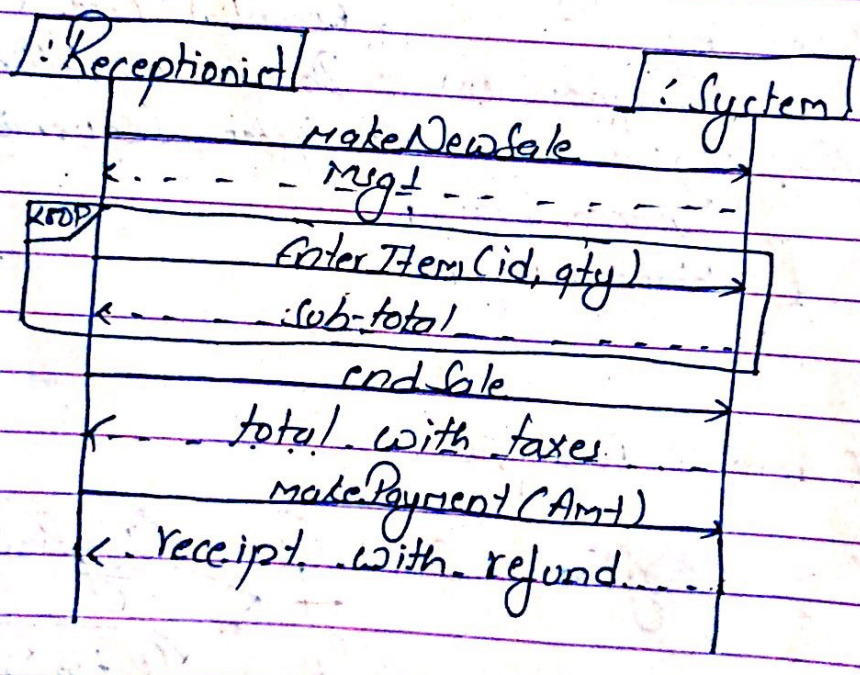


Fig: Domain Modelling

(iii)



55

(#) What is Iterative and Evolutionary Development?

⇒ In iterative approach, development is organized into a series of short, fixed-length mini projects called iteration. Outcome of each iteration is a tested, integrated and executable partial system. Each iteration include its own requirement analysis, design and implementation and testing activities. The system grows incrementally overtime, iteration by iteration, and thus this approach is also known as iterative and incremental development.

Iterative development is embrace change (accept change). Each iteration involves choosing a small subset of the requirements and quickly designing, implementing and testing. In early iterations the choice of requirements and design may not be exactly what is ultimately desired. But the act of swiftly taking a small step, before all requirements are finalized, or the entire design is speculatively defined, leads to rapid feedback

Q. What is GRASP? What are it's principles?

=> GRASP is an acronym for General Responsibility Assignment Software Patterns. GRASP provides guidance for assigning responsibilities to classes and, to a limited extent, determining the classes that will be in an object-oriented system.

In short GRASP stands for designing objects with responsibilities.

The GRASP principles or patterns are a learning aid to help you understand essential object design in a Methodical, rational, explainable way. This approach is based on patterns of assigning responsibilities. GRASP defines 9 basic Object Oriented Design principles or basic building blocks in design.

- -> Information Expert
- -> Creator
- -> Controller
- -> low coupling
- -> High Cohesion
- -> Polymorphism
- -> Pure fabrication
- -> Indirection
- -> Protected variations.

~~GRASP - Examples~~

Information Expert

Expert pattern deals with the problem of what is a general principle of assigning responsibilities to objects? And gives the solution as: Assign a responsibility to the information expert in the class that has the information necessary to fullfill the responsibility. for eg:- Who should be responsible for knowing grand total of sale? And the answer is if there are relevant classes in Design Model, look there first and if not look in Domain Model and attempt to use it's representation to inspire the creation of corresponding design classes.

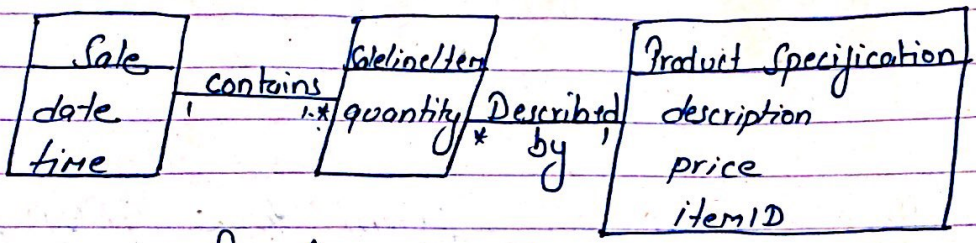


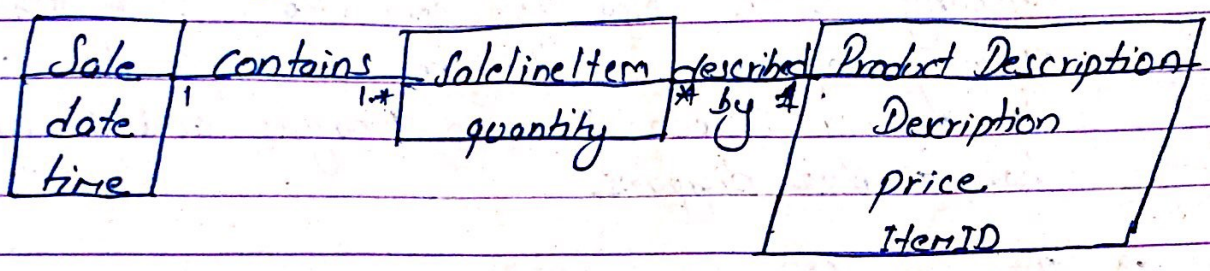
Fig:- Associations for Sale.

Creator pattern

Whereas Creator pattern deals with the problem as who should be responsible for creating a new instance of some class? And gives solution as: Assign class B the responsibility to create an instance of class A if one or more of the following is true:

- B aggregates A objects
- B contains A objects
- B records instances of A objects
- B closely uses A objects
- B has the initializing data that will be passed to A when it is created.

Here, B is creator of A objects. If one or more options apply, prefer a class B which contains class A. for eg:- Who should be responsible for creating a SaleLineItem objects, the creator pattern suggests to choose sale as an instance of SaleLineItem.



Steps for Creating Domain Model

1. List the candidate conceptual classes using the Conceptual Class Category list and noun phrase identification techniques related to the current requirements under consideration.
2. Draw them in a domain model.
3. Add the associations necessary to record relationships for which there is a need to preserve some memory.
4. Add the attributes necessary to fulfill the information requirements.

(22) Low Coupling: How to support low dependency, low change impact, and increased reuse?
⇒ Assign a responsibilities so that coupling remains low.

Coupling is a measure of how strongly one element is connected to., has knowledge of, or relies on other elements. Low coupling is a evaluative pattern that dictates how to assign responsibilities to support:

- lower dependency between the classes
- change in one class having lower impact on other classes.
- higher reuse potential.

A class with high coupling suffer from following undesirable problems:

(i) Changes in related class force local changes
(ii) Harder to understand in isolation.
(iii) Harder to reuse because its use requires the additional presence of the classes on which it is dependent.

Benefits are:-

(i) not affected by changes in other components
(ii) Simple to understand in isolation
(iii) Convinient to reuse.

⊛ High Cohesion:

Q-> How to keep complexity managable?

A=> Assign a responsibility so that cohesion remains high.

- cohesion is a measure of how strongly related and focused the responsibilities of an an elements are.
- An element with highly related responsibilities, and which does not do a tremedous amount of work, has high cohesion.
- A class with low cohesion does many unrelated things or does too much work. Such classes are undesirable; they suffer from following problems:
  - -> Hard to comprehend
  - -> Hard to reuse
  - -> Hard to maintain
  - -> delicate; constantly effected by change.

Controller Pattern.

Problem: Who should be responsible for handling an input system event?

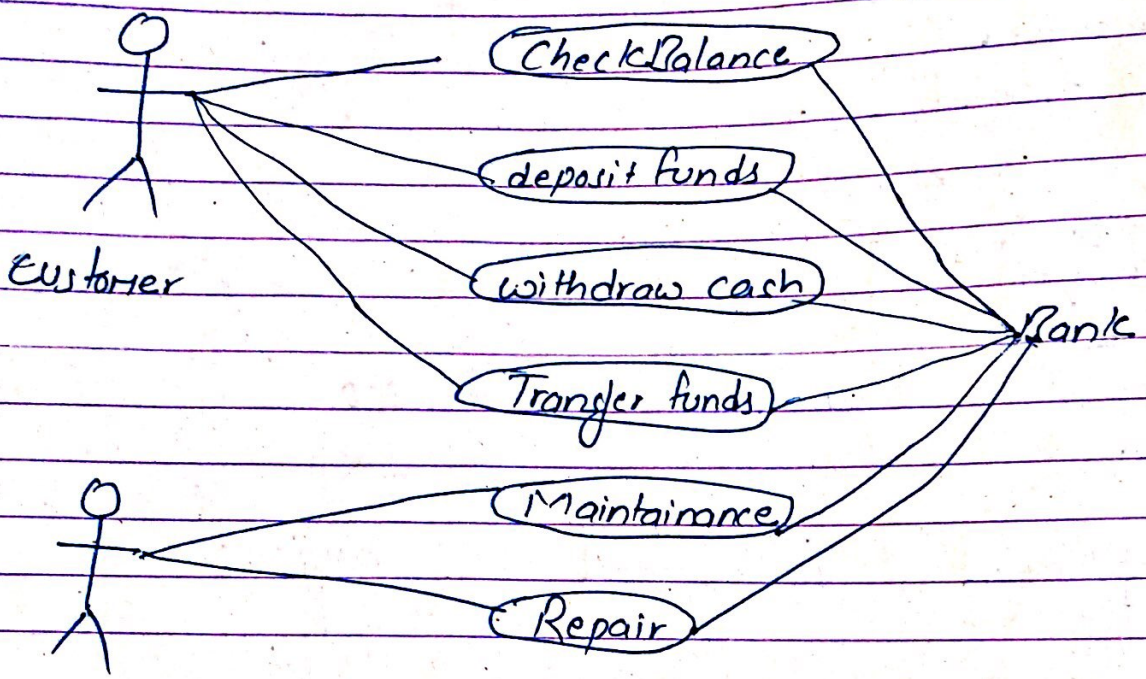
Solution:- An input system event is an event generated by an external actor. They are associated with system operations- operations of the system in response of system events, just as methods are related.

For eg: If cashier presses "endsale" button he is generating a system event indicating "a sale has ended".

⇒ So, assign the responsibility for receiving or handling a system event message to class representing one of the following choices:

⇒ Represents, the overall system, device or subsystem.
⇒ Use the same controller class for all system events in the same use case scenario.

⊛ Use case for ATM transaction:



Static
1. Class diagram
2. Object diagram
3. Package diagram
   ↳ Model diagram
4. Composite Structure
   ↳ 4.1 internal
   ↳ 4.2 Collaboration

Dynamic
⑤ Use Case
⑩ Information
⑪ Activity
⑫ State Machine
   ↳ Behavioural sets
      ↳ problem

Q. How procedure Analysis and Design is different from OOAD, illustrate with examples.

Phase	Structured	Object-oriented.
- Method	SDLC	Iterative / Incremental
- Focus	Process	Objects
- Risk	High	low
- Reuse	low	High
- Maturity	Matured and widespread	Emerging.
- Suitable for	Well-defined projects with stable user requirement	Risky large projects with changing user requirements.
- Analysis	Structuring Requirements: - DFD's - Structured English - Decision Table - ER Analysis	Requirement Engineering: - Use Case Model - Object Model • find Classes & class relation • Object Interaction.

Q. Explain different phases and discipline of Rational Unified Process?

Rational unified process is a complete software development process framework, developed by Rational Corporation. Its an iterative development methodology based upon six industry proven best practices.

RUP is divided into four phases:

i) Inception phase.

ii) Elaboration Phase
iii) Construction Phase
iv) Transition Phase.

i) Inception phase:

The primary objective is to scope the system adequately as a basis for validating initial costing and budgets. To complement the busine case, a basic use case model, project plan, initial risk assesment and project description are generat It is the approximate vicion, business care, scope vovge estimates

ii) Elaboration Phase:

The Primary objective is to mitigate the key risk items identified by analysis up to the end of this phase. The elaboration phase is where the project starts to take shape. In this phase, the problem domain analysis is made and the architecture of the projecto gets it's basic form. It is the resolution of high risks identification of most requirements and scope, more realistic estimates.

iv) Construction Phase:

The primary objective is to build th software system. In this phase, the main focus is on the development of components and other

features of the system.

iv) Transition Phase:

The primary objective is to transit the system from development into production, making it available to users and understood by the end user. The activities of this ~~Beta testing~~ phase include training the end users and Maintainers and beta testing the system to validate it against the end user's expectations.

RUP follows following disciplines:

① Business Modelling:

Discovers all business use cases. Details a single set of business use case.

(I) Requirements:

Discovers all requirement use cases.

(II) Analysis and Design: Decides on technologies for the whole solution.

(IV) Implementation:

Owns the build plans that shows what classes will integrate with one another.

(V) Test:

Ensures that the text is complete and

conducted for the right motivators.

(vi) Deployment
Oversees deployment for all deployment units.

(vii) Project Management:
It makes go/no decisions.

(viii) Environment:
Creates guidelines for using a specific tool.

(ix) Configure and Change Management:
Sets up policies and plans and establishes a change control process.

(i) WhiteBox :- code
(ii) BlackBox :- Test
(iii) Unit Testing :- Part Part
(iv) Integration Testing : 2+2
(v) System Testing.
(vi) Alpha, beta Testing

Phases of RUP