SHRI RAMDEOBABA COLLEGE OF ENGINEERING AND MANAGEMENT, NAGPUR - 440013

DESIGN PATTERNS

V SEMESTER

COURSE COORDINATOR: RINA DAMDOO

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Syllabus for Semester V, B. Tech. Computer Science & Engineering

(Artificial Intelligence & Machine Learning)

Course Code : CAT5004-2 **Course : Design Pattern (Program Elective-I)**

L: 3Hrs, T: 0 Hr, P: 0Hrs, Per Week **Total Credits: 03**

Course Objectives

1. To learn the fundamentals of software design by referring a catalog of design patterns
2. Demonstrate how to use design patterns to address code development issues.
3. Identify the most suitable design pattern to address a given application design problem.
4. Apply design principles (e.g., open-closed, dependency inversion, substitution, etc).
5. Critique code by identifying and refactoring anti-patterns.

Syllabus for Semester V, B. Tech. Computer Science and Engineering (Data Science)

Course Code: CDT5004-2 **Course: Design Patterns**

L:3 Hrs, T:0 Hrs, P:0 Hrs Per Week **Total Credits: 3**

Course Objectives

1. To learn the fundamentals of software design by referring a catalog of design patterns
2. Demonstrate how to use design patterns to address code development issues.
3. Identify the most suitable design pattern to address a given application design problem.
4. Apply design principles (e.g., open-closed, dependency inversion, substitution, etc).
5. Critique code by identifying and refactoring anti-patterns.

Course Outcomes:

On successful completion of the course, students will be able to:

1. Analyze the need and ability of design patterns in the software design process.
2. Implement various solutions for creation of objects, their structure and the interaction between objects.
3. Develop a loosely coupled application using design patterns.
4. Analyze the tradeoffs of applying a design pattern to a given problem.

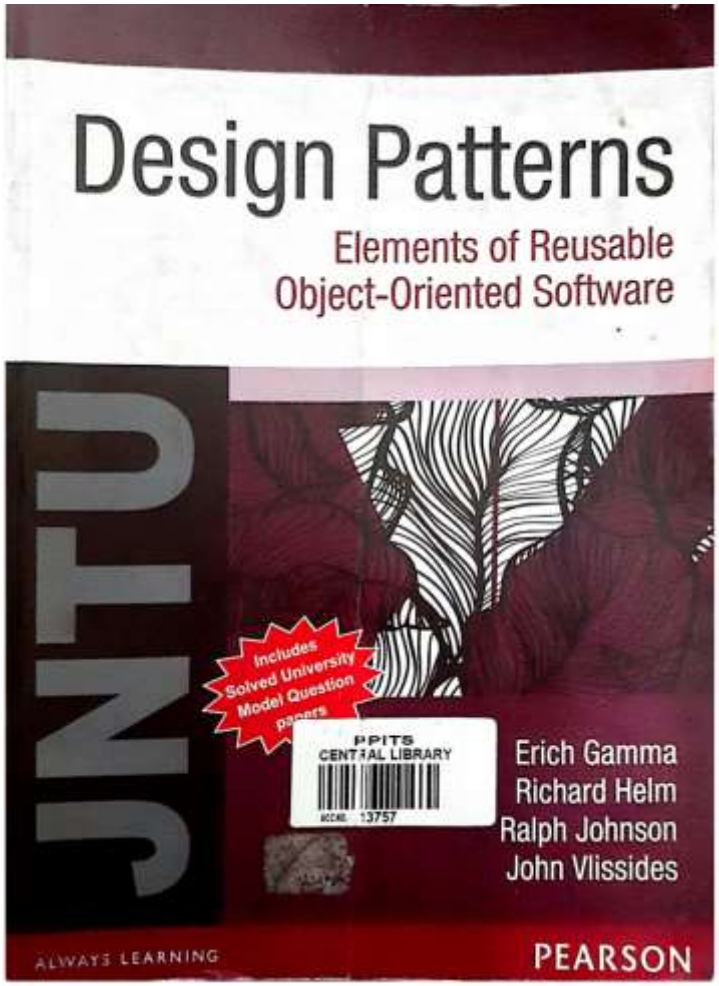
Text Books:

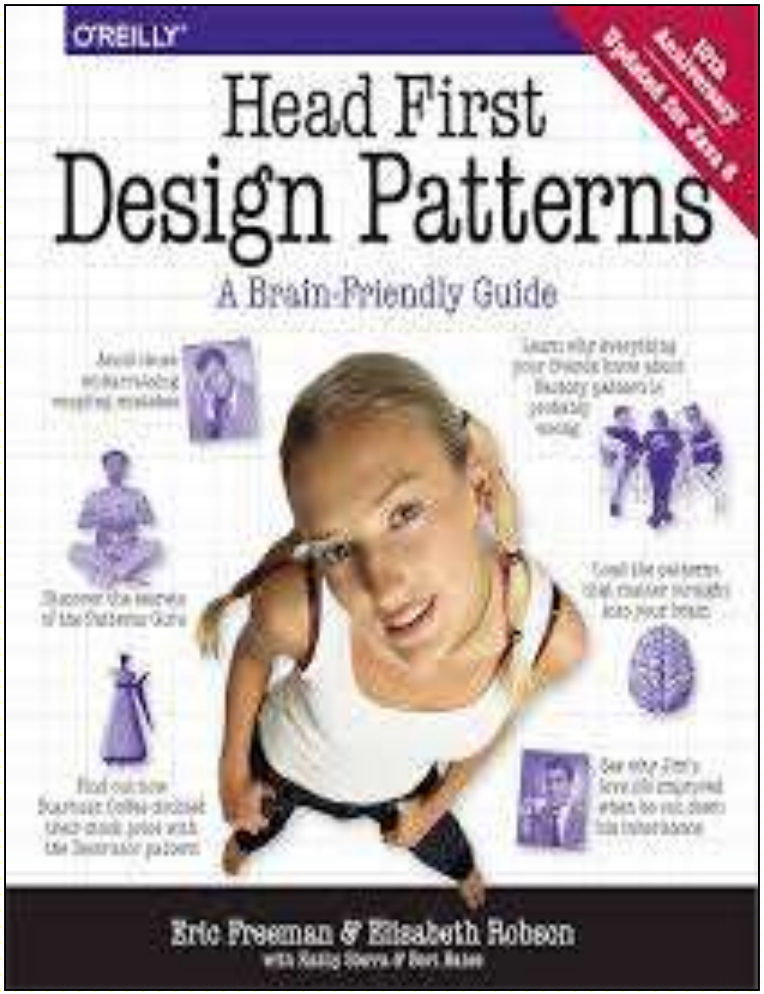
1. Design Patterns: Elements of reusable object-oriented software by Gamma Erich, Helm Richard, Johnson Ralph, and Vlissides John, Pearson Education
2. Design Patterns Explained by Alan Shallowly and James Trott, Addison-Wesley

Reference Books:

1. Pattern's in JAVA Vol-I by Mark Grand, WileyDreamTech.
2. JAVA Enterprise Design Patterns Vol-III by Mark Grand, WileyDreamTech.
3. Head First Design Patterns by Eric Freeman, O'Reilly.

DESIGN PATTERNS TEXT BOOK





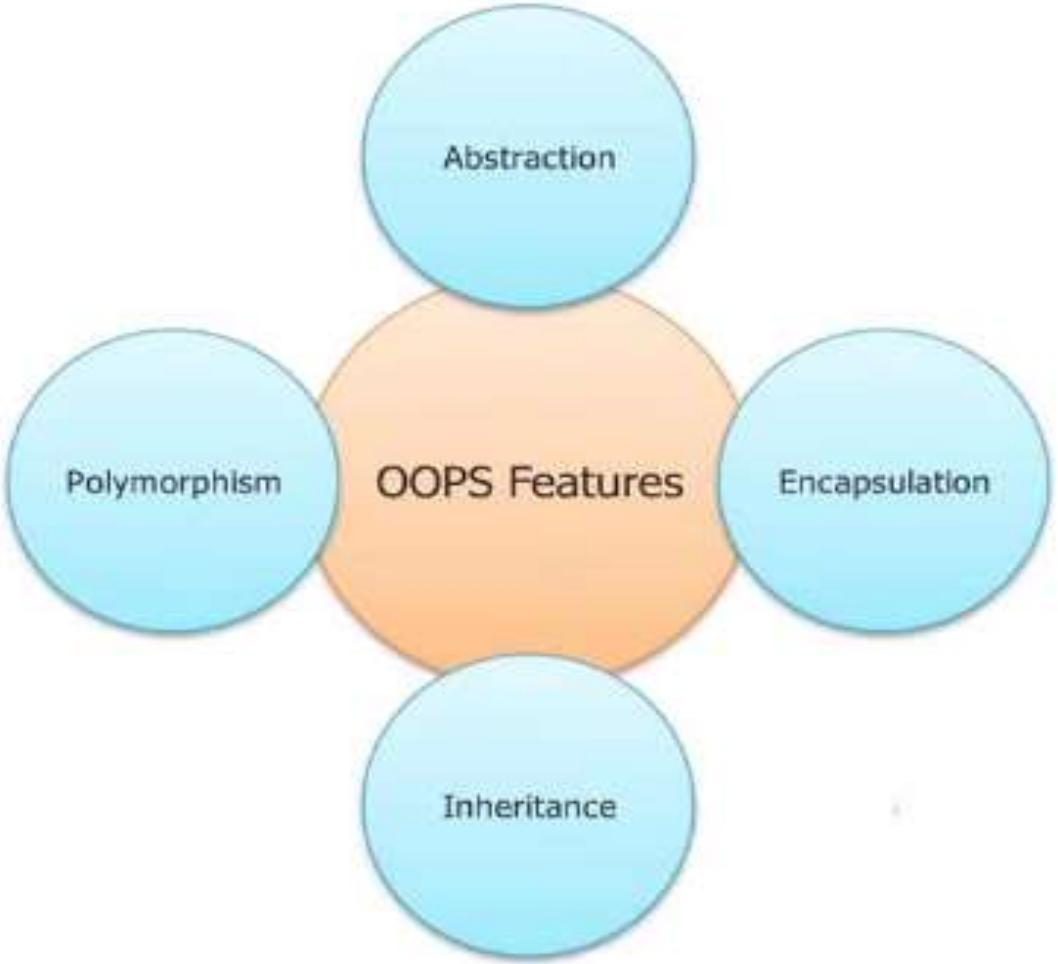
DESIGN PATTERNS UNIT -I

- Introduction to Design Patterns
- Elements of Design Pattern
- Describing Design Pattern
- Classification of Design Patterns
- Role of design Patterns in software design
- How design patterns solve design problems
- Example implementation of design pattern using UML

In object-oriented programming each and every program works with data that describes entities (objects or events) from real life

OBJECT ORIENTED PROGRAMMING PRINCIPLES

OOP enables the easy reuse of code by applying following principles

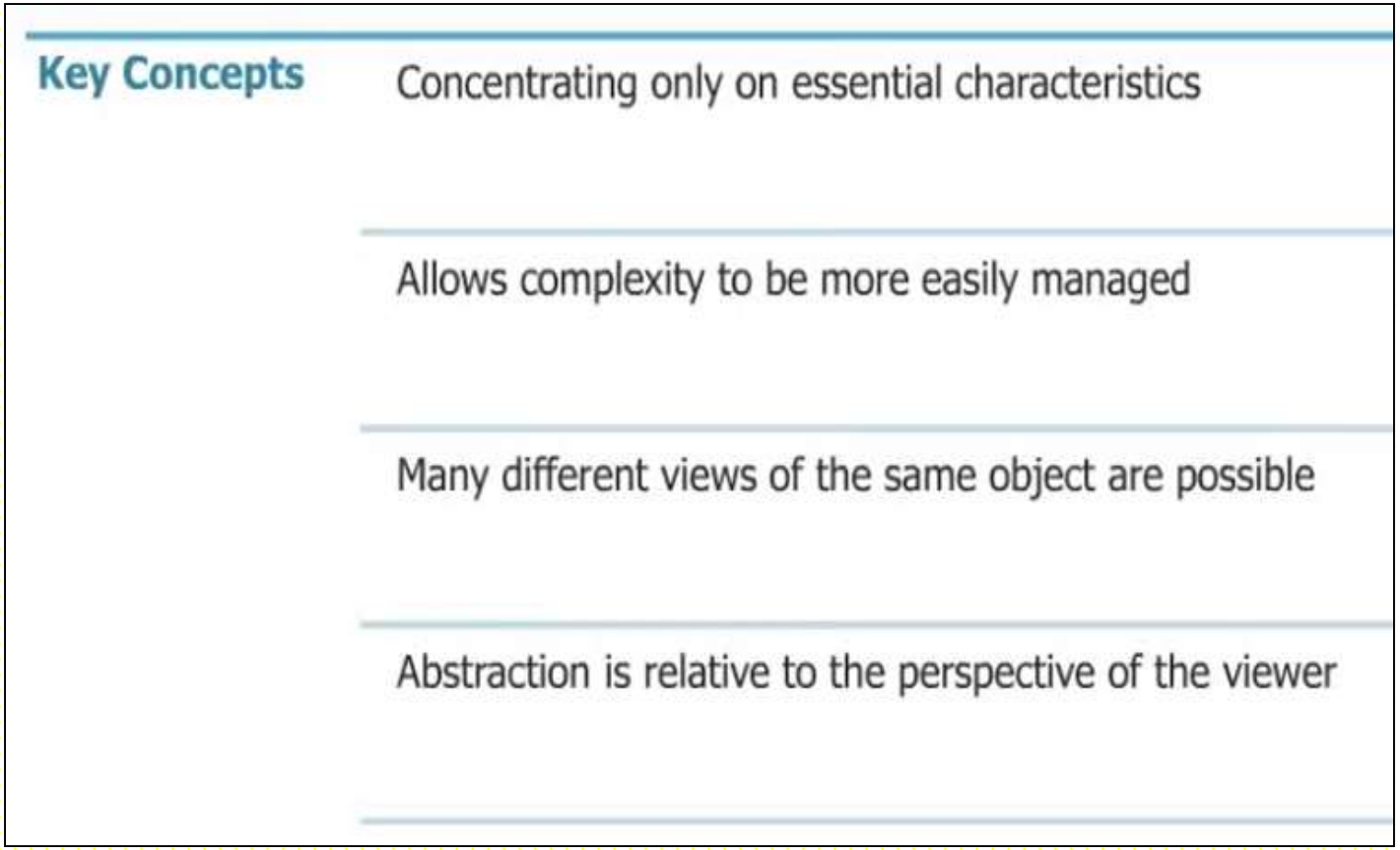


ABSTRACTION

→ The principle of ignoring those aspects of a subject that are not relevant to the current purpose in order to concentrate more on those that are relevant

→ We use abstraction in the development of software to help manage complexity

Abstraction



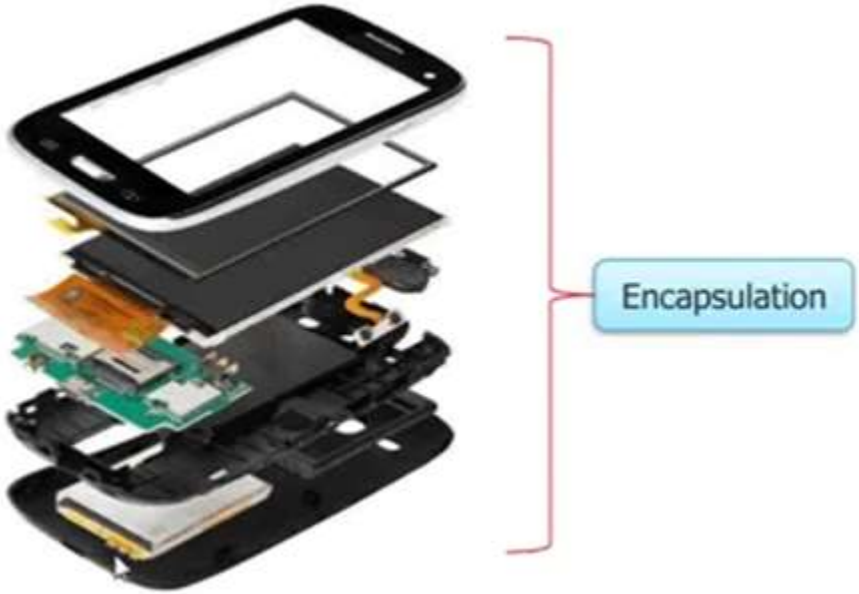
NOTE: Abstraction is achieved by Abstract classes / interfaces

ENCAPSULATION

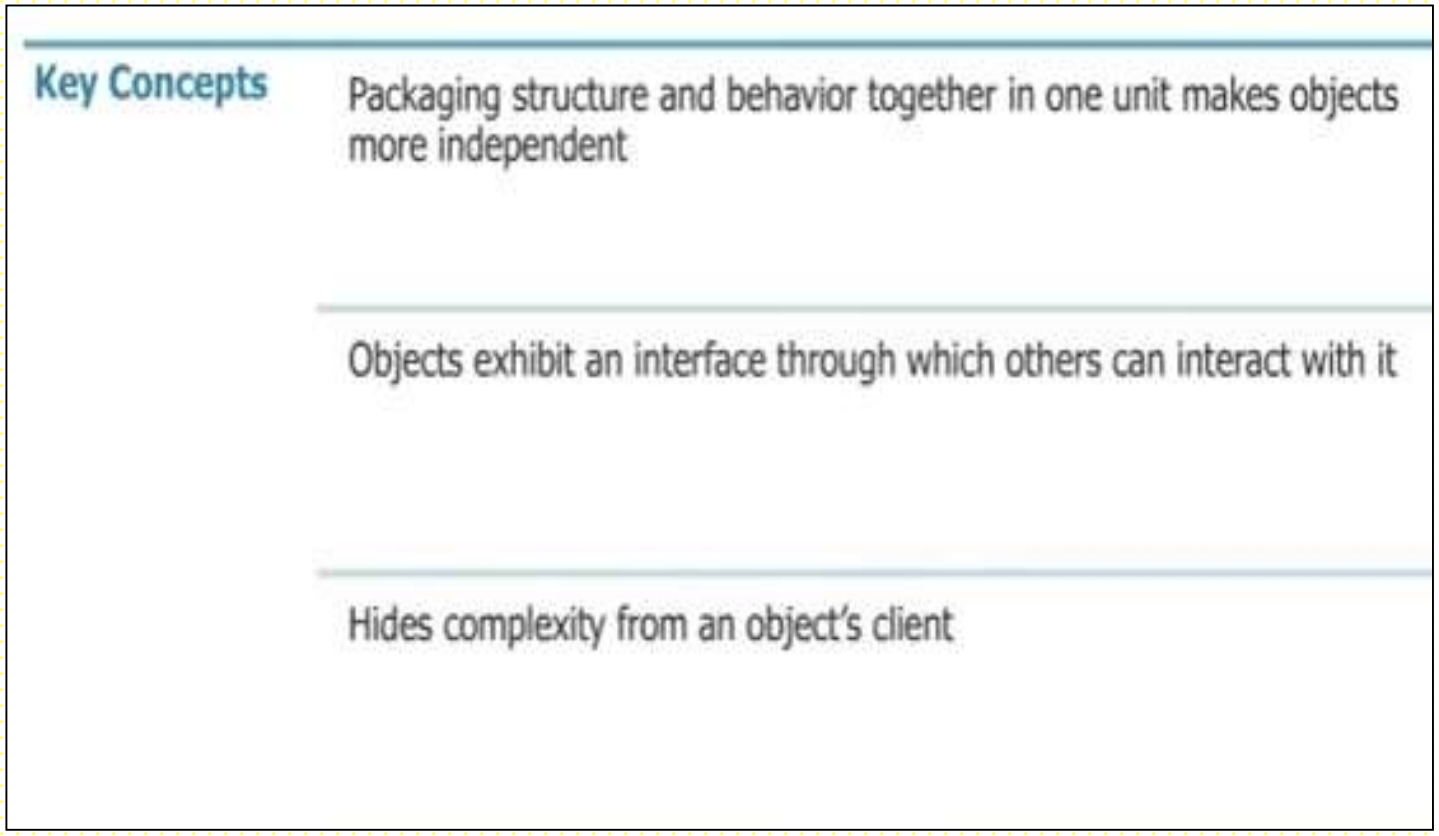
→ The principle of designing a software system in such a manner that the interface to a program component (object) reveals as little as possible about the inner workings of that object

→ Encapsulation (information hiding) helps to minimize rework during maintenance

Encapsulation minimizes maintenance effort



Encapsulation



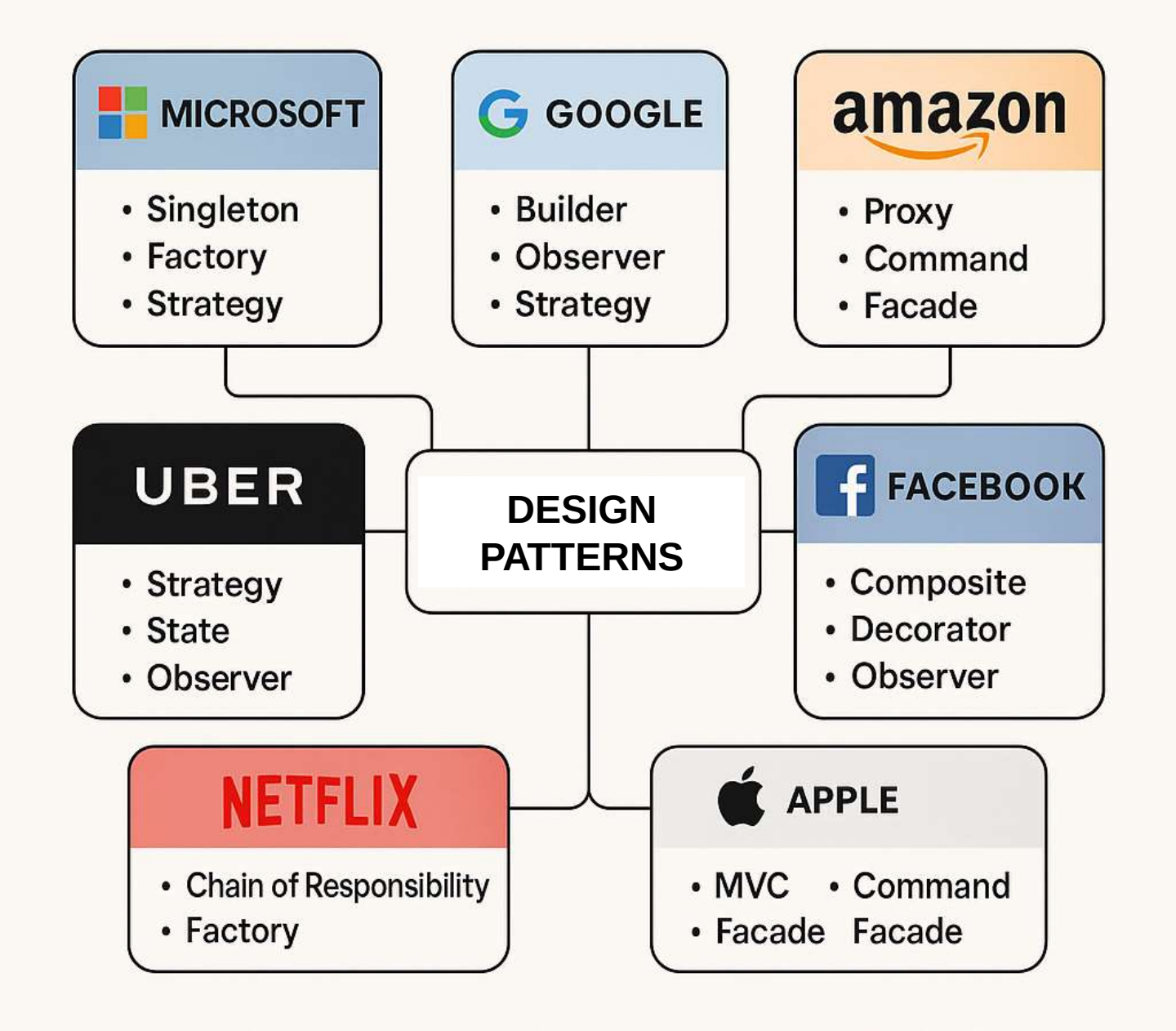
NOTE: Encapsulation is achieved by using private members and public methods.

SOFTWARE DESIGNING

- → Designing is considered to be the most difficult phase of Software Development
- → OOP features alone are not enough to design a software
- → To design a flexible and maintainable software its very important that you understand the architecture of the system

There are well defined software design principles which can be used to create a robust software design

DESIGN PATTERNS



Why these companies use design patterns?

- To improve code maintainability and scalability
- To handle real-time data and complex architectures
- To enforce best practices and reusability

Software Development, Design Patterns, Software Engineering

Feature	Software Development	Design Patterns	Software Engineering
Definition	Writing software code	Reusable design solutions	Structured process to build software
Scope	Code-level implementation	Code structure & architecture	Complete software lifecycle
Focus	Features & logic	Reusability & maintainability	Quality, cost, time, scalability
Example	Writing CRUD app in Python	Applying Factory Pattern	Managing SDLC using Agile

SOLID Principle

→SOLID are five basic principles which help to create good software design

→SOLID is an acronym where:-

- S stands for SRP (**Single Responsibility Principle**)
- O stands for OCP (**Open Closed Principle**)
- L stands for LSP (**Liskov Substitution Principle**)
- I stands for ISP (**Interface Segregation Principle**)
- D stands for DIP (**Dependency Inversion Principle**)

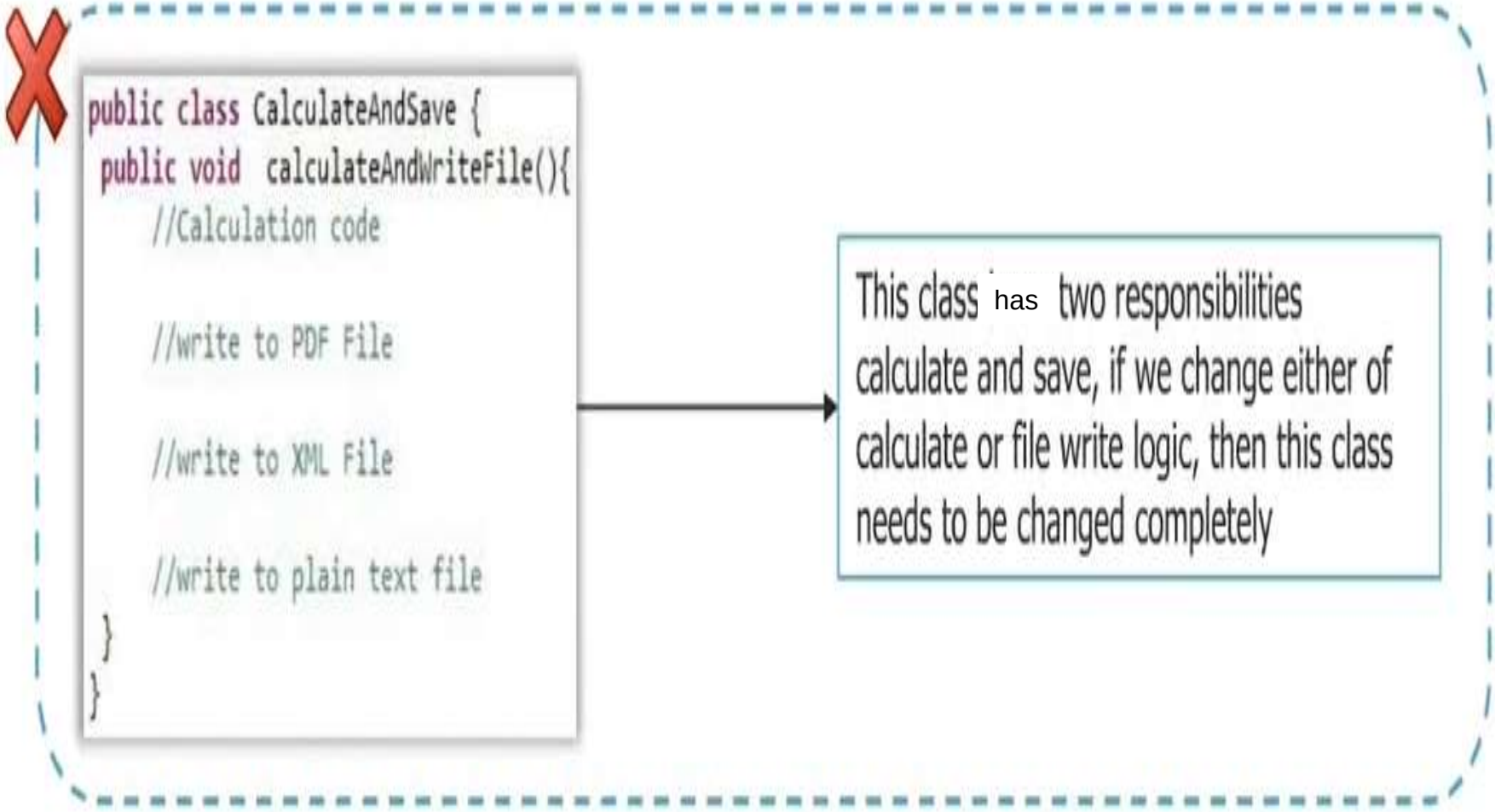
SINGLE RESPONSIBILITY PRINCIPLE

→The Single Responsibility Principle(SRP) states that there should never be more than one reason for a class to change

→This means that every class, or similar structure, in your code should have only one job to do. Everything in the class should be related to that single purpose

→ It does not mean that your classes should only contain one method or property

EXAMPLE


public class CalculateAndSave {
public void calculateAndWriteFile(){
//Calculation code
//write to PDF File
//write to XML File
//write to plain text file
}
}
This class has two responsibilities calculate and save, if we change either of calculate or file write logic, then this class needs to be changed completely


public class Calculate {
public void calculate(){
}
}
public class SavePDF {
public void savePDF(ArrayList<String> data){
}
}
public class SaveXML {
public void saveXML(ArrayList<String> data){
}
}
Responsible for calculation
Responsible for saving as PDF
Responsible for saving as XML

OPEN / CLOSED PRINCIPLE

→The Open/Closed Principle (OCP) states that classes should be open for extension but closed for modification

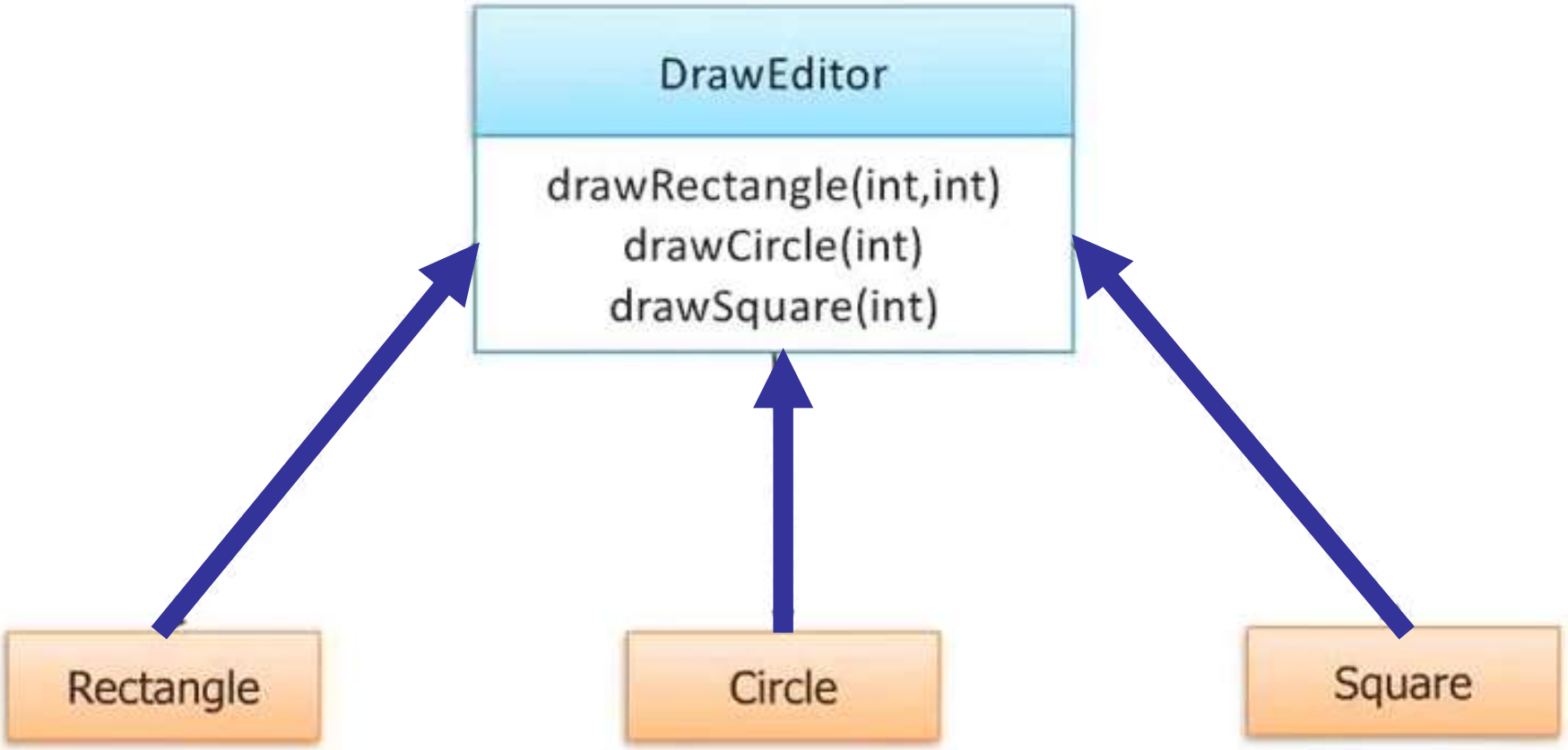
→"Open to extension" means that you should design your classes so that new functionality can be added as new requirements are generated

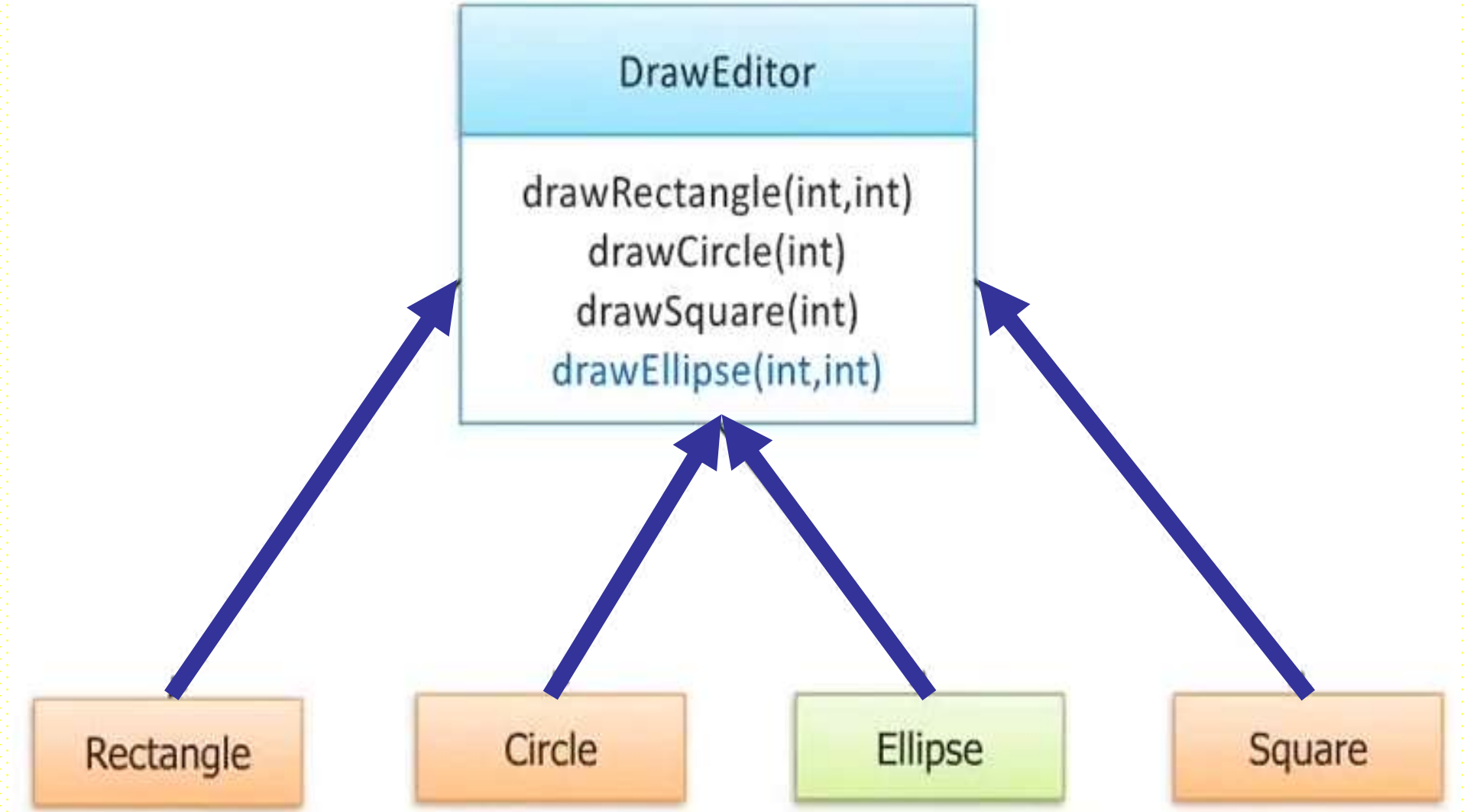
→"Closed for modification" means that once you have developed a class you should never modify it, except to correct bugs

Such an entity can allow its behavior to be extended without modifying its source code

→Consider this scenario we have DrawEditor class with three methods

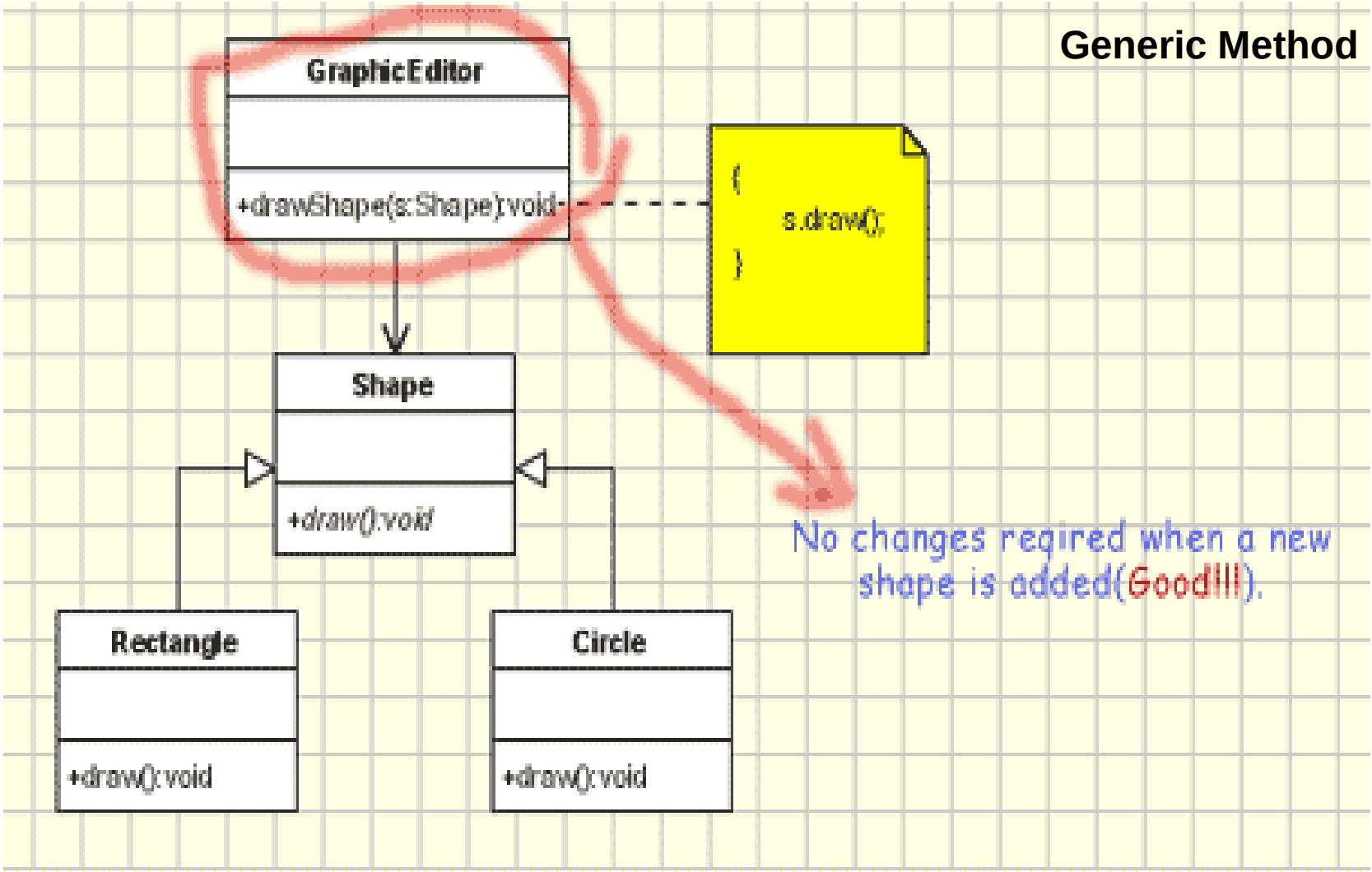
→Suppose we have to add another method to create another shape, then we have to modify the DrawEditor class which violates the Open/Closed principle





DrawEditor class is not closed for modification which is violation of Open/Closed principle

Generic Method



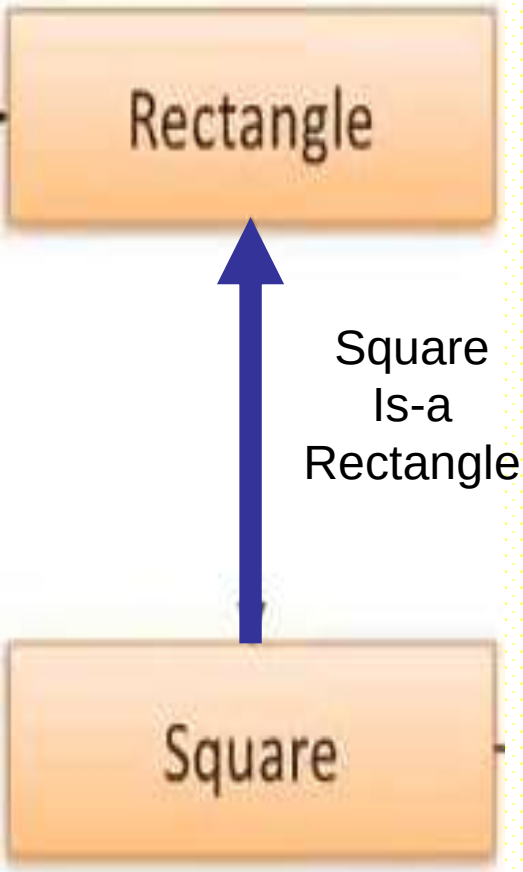
LISKOV SUBSTITUTION PRINCIPLE

→In OOP we create classes then extend some of the classes creating new classes. Sometimes while extending a class the new derived class change the behavior of the base class

→Liskov substitution principle states that derived classes should not change the expected behavior of base class

→In simple terms, if we create a derived class for a base class, it should properly reflect the base class and extend it without replacing the functionality of old classes

```
public class Rectangle {
    float length;
    float breadth;
    public void setLength(float a){
        length=a;
    }
    public void setBreadth(float b){
        breadth=b;
    }
    public float getArea(){
        return length*breadth;
    }
    public float getLength(){
        return length;
    }
    public float getBreadth(){
        return breadth;
    }
}
```




```
public class Square extends Rectangle {

  public void setLength(float a){
      length=a;
      breadth=a;
  }
  public void setBreadth(float b){
      length=b;
      breadth=b;
  }
}
```

```
public class LiskovViolationTest {

  public static Rectangle getRectangle(){
      //This method can return Rectangle or any subclass of it
      return new Square();
  }

  public static void main(String args[]){
      Rectangle rectangle=LiskovViolationTest.getRectangle();
      rectangle.setLength(4);
      rectangle.setBreadth(6);
      System.out.println("Rectangle Area is : "+rectangle.getArea());
  }
}
```

LISKOV SUBSTITUTION PRINCIPLE.....

User thinks he is setting length and breadth for rectangle

user expects area to be 24 but it prints 36

```java
public class Square {
 float side;
 public float getSide(){
    return side;
 }
 public float getArea(){
    return side*side;
 }
}
```

```java
public class Test {
   public static Rectangle getRectangle(){
       // return new Square(); , it will not compile
       return new Rectangle();
   }

   public static void main(String args[]){
       Rectangle rectangle=Test.getRectangle();
       rectangle.setLength(4);
       rectangle.setBreadth(6);
       System.out.println("Rectangle Area is : "+rectangle.getArea());
   }
}
```

INTERFACE SEGREGATION PRINCIPLE

→The Interface Segregation Principle (ISP) states that clients should not be forced to depend upon interfaces that they do not use

→ISP guides us to create multiple, smaller, cohesive interfaces

→The original class implements each interface

→Client code can then refer to the class using the smaller interface without knowing that other members exist

Violating Interface Segregation Principle

```
public interface Animal {
    public void fly();
    public void run();
    public void crawl();
    public void swim();
}
```

```
public class Crane implements Animal {
    @Override
    public void fly() {
        System.out.println("Crane`s fly above 1000 feet");
    }
    @Override
    public void run() {
        // do nothing
    }
    @Override
    public void swim() {
        // do nothing
    }
    @Override
    public void crawl() {
        // do nothing
    }
}
```

Animal interface forces each Animal to implement all the methods

Adhering to Interface Segregation Principle

Segregating interfaces

```
public interface Fly {
 public void fly();
}
```

```
public interface Sound {
 public void makeSound();
}
```

```
public interface Runner {
   public void run();
}
```

```
public interface Crawl {
   public void crawl();
}
```

```java
public class Dog implements Runner,Sound {
    @Override
    public void makeSound() {
        System.out.println("Dog runs at medium speed");
    }
```

```java
public class Crane implements Fly,Sound,Runner {
    @Override
    public void fly() {
        System.out.println("Crane`s fly above 1000 feet");
    }
    @Override
    public void run() {
        System.out.println("Crane runs fast");
    }
    @Override
    public void makeSound() {
        System.out.println("Crane whoops");
    }
```

DEPENDENCY INVERSION PRINCIPLE

→The Dependency Inversion Principle (DIP) states that high level modules should not depend upon low level modules, both should depend upon abstractions

→Secondly, abstractions should not depend upon details

→Details should depend upon abstractions

Dependency Inversion Principle (Contd.)

→ Consider an organization with different level of employees

→ All the low level employee communicated with the Team Lead to pass message to manager

→ Similarly, all the higher level manger communicate the Team Leads to pass information to the employee

→ Here the abstraction level is the Team Lead

Team Lead With All

Team Leads With High Level Manager

Dependency Inversion Principle at Work

→ Lets suppose you have to build an application, where you read data from keyboard and store to a text file

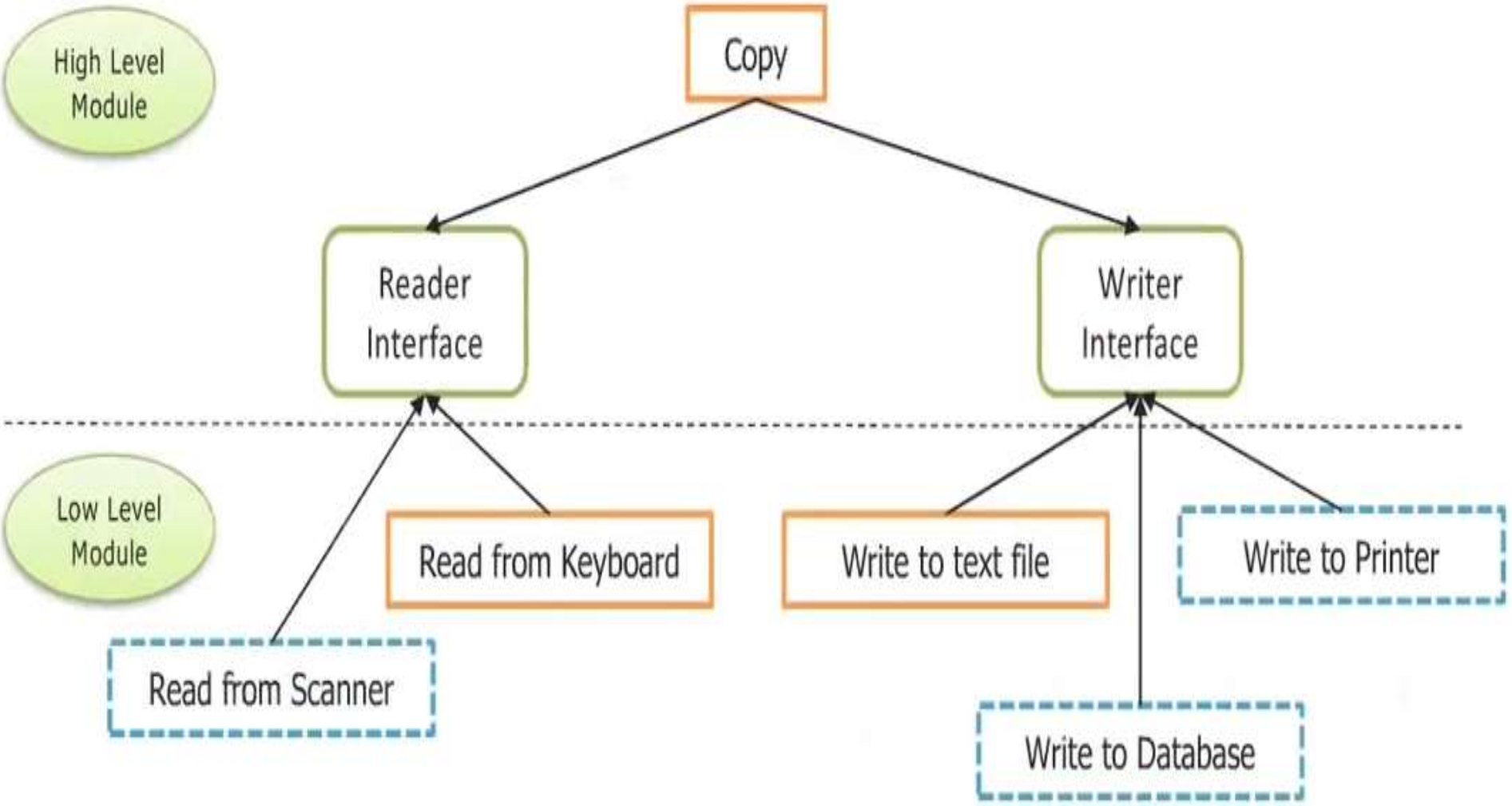
→ Now you have build the application that satisfies the above requirements

→ But now requirements have changed and now you have to take the input data and save it to database

As requirements will change Copy Program will keep changing as it directly depend on low level details like where it is reading from and where it is writing to

→ Now after implementing Dependency Inversion Principle Copy program does not depend upon where its reading from or where its writing

→ Copy program is high level module which depends upon reader and writer interface (abstraction)



TRY THIS .....

Create a hierarchy of Point classes where interface Point has method double length(Point p) and two subclasses –

- 2DPoint(x,y)
- 3DPoint(x,y,z).

Create a class having a factory method that returns a Point depending on the input arguments passed.
Provide the class diagram for your solution and write main program to invoke the factory method twice. Once for a 2DPoint and once for 3DPoint.
Also find the length between those two points.

DESIGN PATTERNS

→In software engineering, a design pattern is a general repeatable solution to a commonly occurring problem in software design

→A design pattern isn't 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

→Design patterns can speed up the development process by providing tested, proven development paradigms

→Effective software design requires considering issues that may not become visible until later in the implementation

DESIGN PATTERNS

- ✓ Your design should be specific to the problem at hand but also general enough to address future problems and requirements. You also want to avoid redesign.
- ✓ One thing expert designers know, NOT TO DO is ‘solve every problem from scratch’. Rather, they reuse solutions that have worked for them in the past. When they find a good solution, they use it again and again.
- ✓ Consequently, you'll find recurring patterns of classes and communicating objects in many object-oriented systems.
- ✓ These patterns solve specific design problems and make object-oriented designs more flexible, elegant, and ultimately reusable.
- ✓ They help designers reuse successful designs by basing new designs on prior experience. A designer who is familiar with such patterns can apply them immediately to design problems without having to rediscover them.

Elements of Design Pattern

1) Pattern name:
   - The pattern name is a handle we can use to describe a design problem, its solutions, and consequences in a word or two.
   - Naming a pattern immediately increases our design vocabulary.
   - It lets us design at a higher level of abstraction.

2) Problem:
   - The problem describes when to apply the pattern.
   - It explains the problem and its context.
   - It might describe specific design problems such as how to represent algorithms as objects.
   - It might describe class or object structures that are symptomatic of an inflexible design.
   - Sometimes the problem will include a list of conditions that must be met before it makes sense to apply the pattern.

Elements of Design Pattern

3) Solution:

- The solution describes the elements that make up the design, their relationships, responsibilities, and collaborations.
- The solution doesn't describe a particular concrete design or implementation, because a pattern is like a template that can be applied in many different situations. Instead, the pattern provides an abstract description of a design problem and how a general arrangement of elements (classes and objects in our case) solves it.

4) Consequences:

- The consequences are the results and trade-offs of applying the pattern.
- Consequences are critical for evaluating design alternatives and for understanding the costs and benefits of applying the pattern.
- The consequences for software often concern space and time trade-offs. They may address language and implementation issues as well.
- Since reuse is often a factor in object-oriented design, the consequences of a pattern include its impact on a system's flexibility, extensibility, or portability.

Describing Design Patterns...

Pattern Name and Classification

The pattern's name conveys the essence of the pattern succinctly. A good name is vital, because it will become part of your design vocabulary.

Intent

A short statement that answers the following questions: What does the design pattern do? What is its rationale and intent? What particular design issue or problem does it address?

Also Known As

Other well-known names for the pattern, if any.

Motivation

A scenario that illustrates a design problem and how the class and object structures in the pattern solve the problem. The scenario will help you understand the more abstract description of the pattern that follows.

Describing Design Patterns...

Applicability

What are the situations in which the design pattern can be applied? What are examples of poor designs that the pattern can address? How can you recognize these situations?

Structure

A graphical representation of the classes in the pattern using a notation based on the Object Modeling Technique (OMT).

Participants

The classes and/or objects participating in the design pattern and their responsibilities.

Collaborations

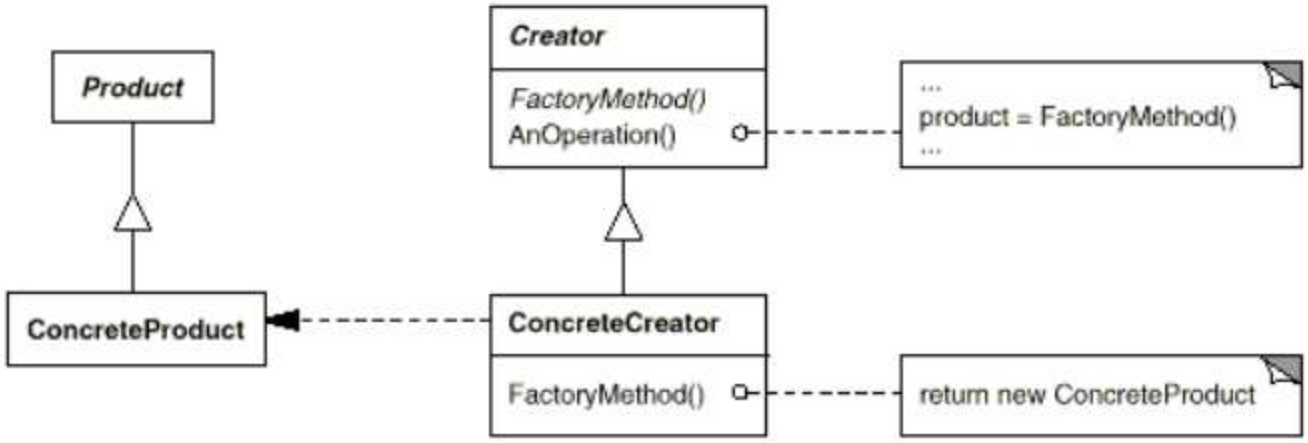
How the participants collaborate to carry out their responsibilities.

Applicability

Use the Factory Method pattern when

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

Structure



Participants

- **Product** (Document)
  - defines the interface of objects the factory method creates.
- **ConcreteProduct** (MyDocument)
  - implements the Product interface.
- **Creator** (Application)

Describing Design Patterns...

Consequences

How does the pattern support its objectives? What are the trade-offs and results of using the pattern? What aspect of system structure does it let you vary independently?

Implementation

What pitfalls, hints, or techniques should you be aware of when implementing the pattern? Are there language-specific issues?

Sample Code

Code fragments that illustrate how you might implement the pattern in C++ or Smalltalk.

Known Uses

Examples of the pattern found in real systems. We include at least two examples from different domains.

Related Patterns

What design patterns are closely related to this one? What are the important differences? With which other patterns should this one be used?

Classification of Design Patterns

We classify design patterns by two criteria:

- Purpose
- Scope

Patterns by purpose can be:

- Creational Design Patterns
- Structural Design Patterns
- Behavioral Design Patterns purpose

Patterns by Scope can be:

- Class Design Patterns
- Object Design Patterns

Classification of Design Patterns

		By Purpose		
		Creational	**Structural**	**Behavioral**
By Scope	**Class**	• Factory Method	• Adapter (class)	• Interpreter • Template Method
	Object	• Abstract Factory • Builder • Prototype • Singleton	• Adapter (object) • Bridge • Composite • Decorator • Façade • Flyweight • Proxy	• Chain of Responsibility • Command • Iterator • Mediator • Memento • Observer • State • Strategy • Visitor

Classification of Design Patterns (Purpose)

Creational class patterns defer some part of object creation to subclasses, while Creational object patterns defer it to another object.

The Structural class patterns use inheritance to compose classes, while the Structural object patterns describe ways to assemble objects.

The Behavioral class patterns use inheritance to describe algorithms and flow of control, whereas the Behavioral object patterns describe how a group of objects cooperate to perform a task that no single object can carry out alone.

Classification of Design Patterns (Scope)

- The second criterion, called scope, specifies whether the pattern applies primarily to classes or to objects.
- Class patterns deal with relationships between classes and their subclasses. These relationships are established through inheritance, so they are static— fixed at compile-time.
- Object patterns deal with object relationships, which can be changed at run-time and are more dynamic.
- Almost all patterns use inheritance to some extent. So the only patterns labeled "class patterns" are those that focus on class relationships.

Creational Design Patterns

→These design patterns are all about class instantiation

→This pattern can be further divided into class-creation patterns and object-creational patterns

→While class-creation patterns use inheritance effectively in the instantiation process, object-creation patterns use delegation effectively to get the job done

- » Abstract Factory
- » Builder
- » Factory
- » Prototype
- » Singleton

Structural Design Patterns

→These design patterns are all about Class and Object composition

→Structural class-creation patterns use inheritance to compose interfaces

→Structural object-patterns define ways to compose objects to obtain new functionality

- » Adapter
- » Bridge
- » Composite
- » Decorator
- » Facade
- » Flyweight
- » Proxy

Behavioral Design Patterns

→These design patterns are all about Class's objects communication

→Behavioral patterns are those patterns that are most specifically concerned with communication between objects

» Chain of Responsibility

» Command

» Iterator

» Mediator

» Observer

» Strategy

» Template method

» Visitor