**Introduction**

Designing object-oriented software is hard, and designing reusable object-oriented software is even harder.

You must find appropriate objects, factor them into classes at the right granularity, define class interfaces and inheritance hierarchies, and establish key relationships among them.

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, or at least minimize it.

Experienced object-oriented designers will tell you that a reusable and flexible design is difficult if not impossible to get "right" the first time. Before a design is finished, they usually try to reuse it several times, modifying it each time.

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.

In this unit we will study some of the tested Patterns which have been designed by experts and which can make the our design Flexible and Reusable.

**What is a Design Pattern?**

Christopher Alexander says, "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of 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".

In general, a pattern has four essential elements:

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.
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. Though consequences are often unvoiced when we describe design decisions, they 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.

**NEED OF DESIGN PATTERNS**

* Design Patterns are already defined and provides industry standard approach to solve a recurring problem, so it saves time if we sensibly use the design pattern.
* Using design patterns promotes reusability that leads to more robust and highly maintainable code. It helps in reducing total cost of ownership (TCO) of the software product.
* Since design patterns are already defined, it makes our code easy to understand and debug. It leads to faster development and new members of team understand it easily.
* Families of related patterns help to programmer to learn patterns faster and can direct efforts to find new patterns.

**ADVANTAGE OF DESIGN PATTERN:**

1. They are reusable in multiple projects.

2. They provide the solutions that help to define the system architecture.

3. They capture the software engineering experiences.

4. They provide transparency to the design of an application.

5. They are well-proved and testified solutions since they have been built upon the knowledge and experience of expert software developers.

6. Designs patterns donate guarantee an absolute solution to a problem. They provide clarity to the system architecture and the possibility of building a better system.

**WHEN SHOULD WE USE THE DESIGN PATTERNS?**

We must use the design patterns during the analysis and requirement phase of SDLC(Software Development Life Cycle).Design patterns ease the analysis and requirement phase of SDLC by providing information based on prior hands-on experiences.

**WHAT ARE DIFFERENT CHARACTERISTICS OF DESIGN PATTERNS?**

A design pattern is a general repeatable solution to a commonly occurring problem in software design

- It is a proven solution to problems that keep recurring.

- They are reusable solutions to common problems.

- Design patterns are not frameworks.

- Design patterns are more abstract than frameworks.

- Design pattern cannot be directly implemented.

- Design patterns are more primitive than a framework.

- A design pattern cannot incorporate a framework.

- Patterns may be documented using one of several alternative templates.

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

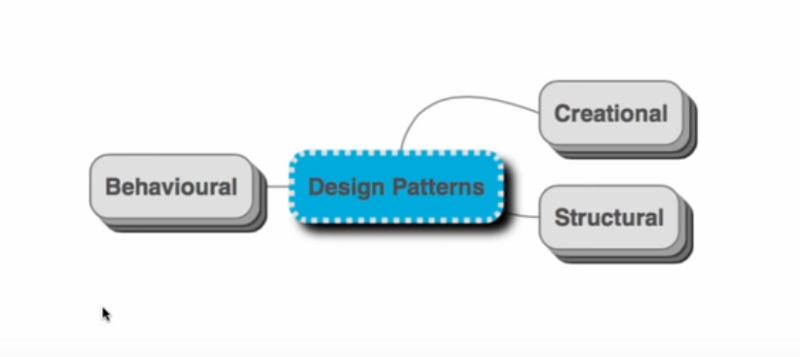
- Reusing design patterns helps to prevent subtle issues that can cause major problems and improves code readability for coders and architects familiar with the patterns.

- Patterns allow developers to communicate using well-known, well understood names for software interactions

***CLASSIFICATION***

We will classify design patterns in 3 categories:

1. **Creational patterns** concern the process of object creation.
2. **Structural patterns** deal with the composition of classes or objects
3. **Behavioral patterns** characterize the ways in which classes or objects interact and distribute responsibility



***Creational Pattern:***

1. Creational design patterns abstract the instantiation process.
2. They help make a system independent of how its objects are created, composed, and represented.
3. A class creational pattern uses inheritance to vary the class that's instantiated, whereas an object creational pattern will delegate instantiation to another object.

Types of Creation Design Patterns

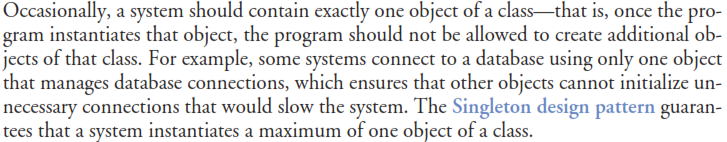
1. Builder Design Pattern
2. Prototype design pattern
3. [Abstract Factory Pattern](https://howtodoinjava.com/design-patterns/creational/abstract-factory-pattern-in-java/)
4. Java Factory Pattern
5. Singleton Pattern

***When to use Creational Pattern:***

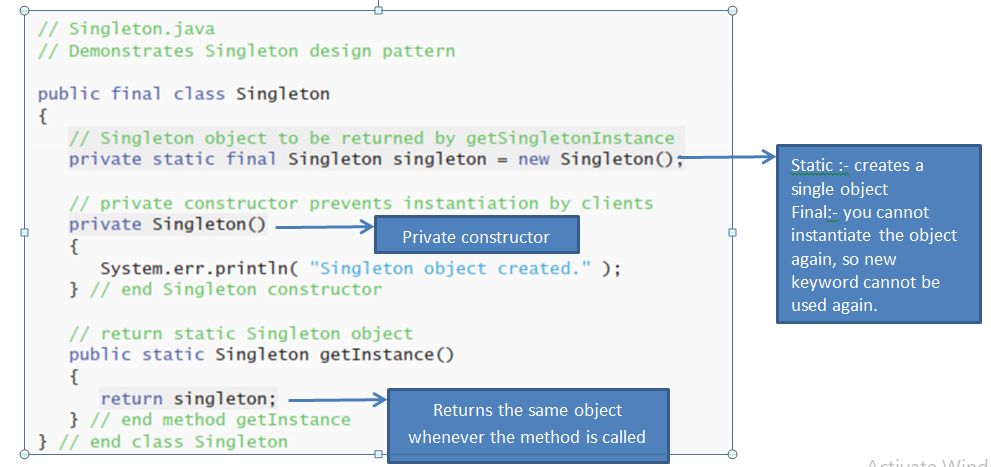
1. A system should be independent of how its objects and products are created.
2. A set of related objects is designed to be used together.
3. Hiding implementation of a class library or product and revealing only their interfaces.
4. Constructing different representations of the independent complex objects.  
   A class wants its subclass to implement the object it creates.

***Singleton Design Pattern:***

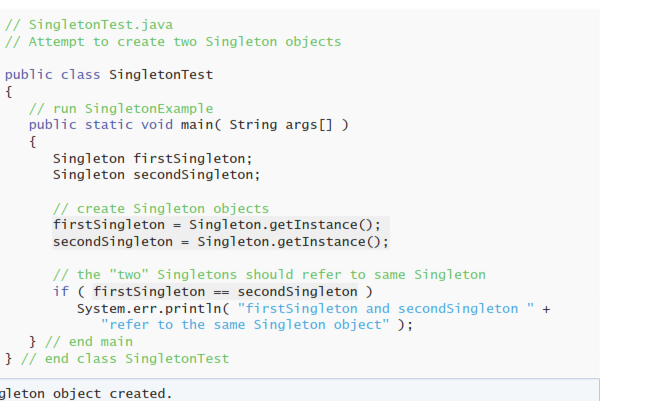
**Singleton pattern** is a design solution where an application wants to have **one and only one instance of any class**, in all possible scenarios without any exceptional condition.



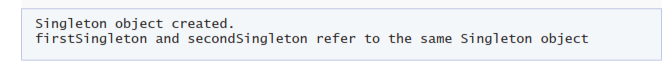
***Singleton Example***



***Using the singleton class***



***OUTPUT:***



***Structural Design Pattern***

1. Structural patterns are concerned with how classes and objects are composed to form larger structures.
2. The structural design pattern **simplifies the structure by identifying the relationships**. These patterns focus on, how the classes inherit from each other and how they are composed from other classes.
3. Structural class patterns use inheritance to compose interfaces or implementations. As a simple example, consider how multiple inheritance mixes two or more classes into one. The result is a class that combines the properties of its parent classes. This pattern is particularly useful for making independently developed class libraries work together.

***Types of Structural patterns***

#### [Adapter Pattern](https://www.javatpoint.com/adapter-pattern)

#### [Bridge Pattern](https://www.javatpoint.com/bridge-pattern)

#### [Composite Pattern](https://www.javatpoint.com/composite-pattern)

#### [Decorator Pattern](https://www.javatpoint.com/decorator-pattern)

#### [Facade Pattern](https://www.javatpoint.com/facade-pattern)

#### [Flyweight Pattern](https://www.javatpoint.com/flyweight-pattern)

#### [Proxy Pattern](https://www.javatpoint.com/proxy-pattern)

***Bridge Pattern***

**Bridge** is a structural design pattern that divides business logic or huge class into separate class hierarchies that can be developed independently.

One of these hierarchies (often called the Abstraction) will get a reference to an object of the second hierarchy (Implementation). The abstraction will be able to delegate some (sometimes, most) of its calls to the implementations object. Since all implementations will have a common interface, they’d be interchangeable inside the abstraction.

***Consider the example***

|  |  |
| --- | --- |
| ***WITHOUT BRIDGE PATTERN***  https://media.geeksforgeeks.org/wp-content/cdn-uploads/im2.png | ***WITH BRIDGE PATTERN***  https://media.geeksforgeeks.org/wp-content/uploads/BridgeDesign3.png |

***Sample code for Bridge Pattern***

// abstraction in bridge pattern

***abstract*** class Vehicle {

protected Workshop workShop1;

protected Workshop workShop2;

protected Vehicle(Workshop workShop1, Workshop workShop2)

{

this.workShop1 = workShop1;

this.workShop2 = workShop2;

}

***abstract public void manufacture();***

}

// Refine abstraction 1 in bridge pattern

***class Car extends Vehicle*** {

public Car(Workshop workShop1, Workshop workShop2)

{

super(workShop1, workShop2);

}

@Override

***public void manufacture() //Give the definition here***

{

System.out.print("Car ");

workShop1.work();

workShop2.work();

}

}

// Refine abstraction 2 in bridge pattern

***class Bike extends Vehicle {***

public Bike(Workshop workShop1, Workshop workShop2)

{

super(workShop1, workShop2);

}

@Override

***public void manufacture()***

{

System.out.print("Bike ");

workShop1.work();

workShop2.work();

}

}

***// Implementor for bridge pattern***

interface Workshop

{

abstract public void work();

}

***// Concrete implementation 1 for bridge pattern***

class Produce implements Workshop {

@Override

public void work()

{

System.out.print("Produced");

}

}

***// Concrete implementation 2 for bridge pattern***

class Assemble implements Workshop {

@Override

public void work()

{

System.out.print(" And");

System.out.println(" Assembled.");

}

}

// Demonstration of bridge design pattern

class BridgePattern {

public static void main(String[] args)

{

Vehicle vehicle1 = new Car(new Produce(), new Assemble());

vehicle1.manufacture();

Vehicle vehicle2 = new Bike(new Produce(), new Assemble());

vehicle2.manufacture();

}

}

#### Advantage of Bridge Pattern

* It enables the separation of implementation from the interface.
* It improves the extensibility.
* It allows the hiding of implementation details from the client.

#### Usage of Bridge Pattern

* When you don't want a permanent binding between the functional abstraction and its implementation.
* When both the functional abstraction and its implementation need to extended using sub-classes.
* It is mostly used in those places where changes made in the implementation does not affect the clients.

***Behavioral Design Pattern***

1. Behavioral patterns are concerned with algorithms and the assignment of responsibilities between objects.
2. Behavioral patterns describe not just patterns of objects or classes but also the patterns of communication between them.
3. These patterns characterize complex control flow that's difficult to follow at run-time. They shift focus away from flow of control to let you concentrate just on the way objects are interconnected.
4. Behavioral object patterns use object composition rather than inheritance

***Types of Behavioral Design Pattern***

1. Chain of Responsibility Pattern
2. Command Pattern
3. Interpreter Pattern
4. Iterator Pattern
5. Mediator Pattern
6. Memento Pattern
7. Observer Pattern
8. State Pattern
9. Strategy Pattern
10. Template Pattern
11. Visitor Pattern
12. Null Object

***Chain of Responsibility Design Pattern***

In chain of responsibility, sender sends a request to a chain of objects. The request can be handled by any object in the chain.

A Chain of Responsibility Pattern says that just **"avoid coupling the sender of a request to its receiver by giving multiple objects a chance to handle the request".** For example, in leave management example the request is granted by the various authorities dependent on number of leaves to be sanctioned.

***CONSIDER THE EXAMPLE***

Consider a leave request is applied and its approval is done by different roles based on number of days.

* If 5 days, then it is approved by Team Leader.
* If 5 to 15 days, then it is approved by Manager.
* If more than 15 days, then it is approved by Owner.

public class ***LeaveRequest*** {

private int days;

  public LeaveRequest(int days) {

this.days = days;

}

public int getDays() {

return days;

}

public void setDays(int days) {

this.days = days;

}

}

***public abstract class Approver*** {

protected Approver nextApprover;

public void approveRequest(LeaveRequest request){

nextApprover.approveRequest(request);

}

public void setNext(Approver approver){

this.nextApprover = approver;

}

}

public class ***TeamLeader extends Approver*** {

@Override

public void approveRequest(LeaveRequest request) {

  if (request.getDays() <= 5) {

System.out.println("Leave request approved for " + request.getDays() + " days by Supervisor");

}

else {

super.approveRequest(request);

}

}

}

public class ***Manager extends Approver*** {

@Override

public void approveRequest(LeaveRequest request) {

if (request.getDays() <= 15 && request.getDays() > 5) {

System.out.println("Leave request approved for " + request.getDays() + " days by Manager");

}

else {

super.approveRequest(request);

}

}

}

public class ***AccountHead extends Approver*** {

@Override

public void approveRequest(LeaveRequest request) {

if (request.getDays() > 15) {

System.out.println("Leave request approved for " + request.getDays() + " days by AccountHead");

}

else {

super.approveRequest(request);

}

}

 }

public class LeaveSystem {

public static void main(String[] args)

{

Approver teamLead = new TeamLeader();

Approver manager = new Manager();

Approver accountHead = new AccountHead();

teamLead.setNext(manager);

manager.setNext(accountHead);

teamLead.approveRequest(new LeaveRequest(4));

teamLead.approveRequest(new LeaveRequest(40));

teamLead.approveRequest(new LeaveRequest(12));

}

}

Points to notice:

* **LeaveRequest** is the actual request object that will be created by client and passed to chain of responsibilities.
* **Approver**is an abstract class which has protected variable **nextApprover,**which will hold reference to next responsibility handler in the chain.
* **TeamLeader**, **Manager** and **AccountHead** are concrete subclasses of **Approver,**which provide their own specific behaviour based on our conditions for leave approval. If they can’t handle **LeaveRequest,**they simply call **super.approveRequest(request),**which in turn calls method from **nextApprover.**
* **LeaveSystem**is actual client which instantiates all responsibilities and create a chain by setting next approver for each responsibility.

#### Advantage of Chain of Responsibility Pattern

* It reduces the coupling.
* It adds flexibility while assigning the responsibilities to objects.
* It allows a set of classes to act as one; events produced in one class can be sent to other handler classes with the help of composition.

#### Usage of Chain of Responsibility Pattern:

It is used:

* When more than one object can handle a request and the handler is unknown.
* When the group of objects that can handle the request must be specified in dynamic way.