Module-2 NOTES

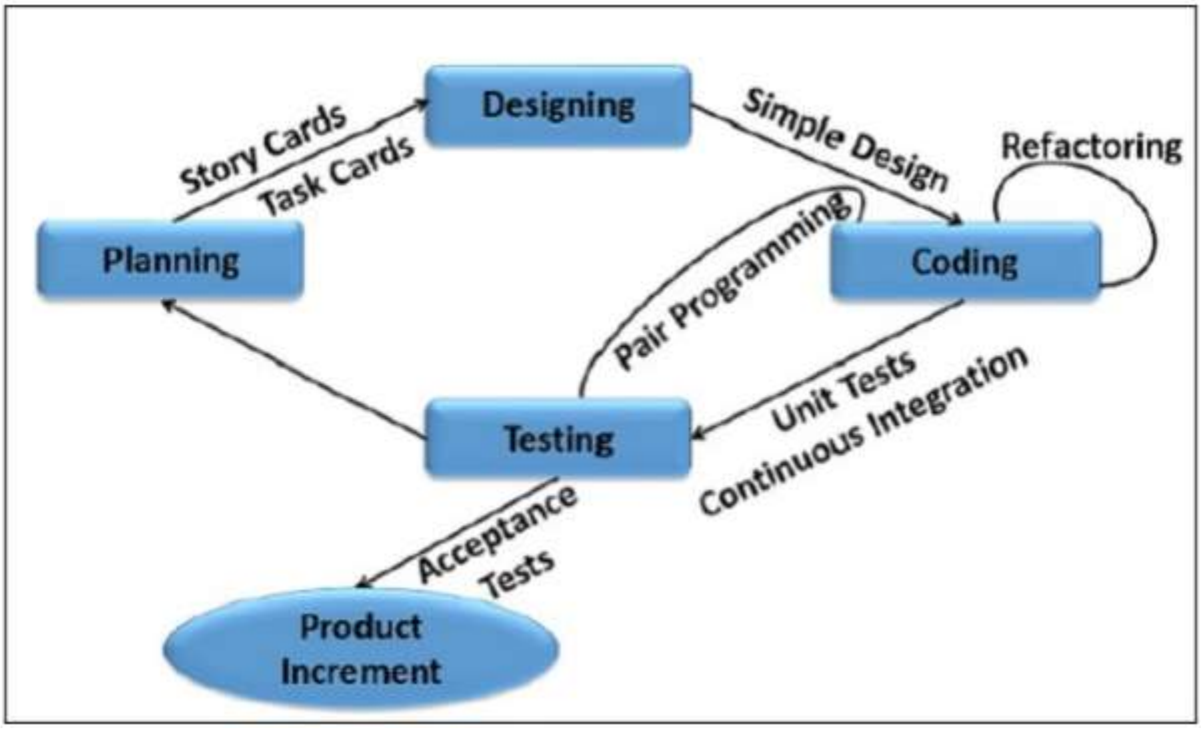
Principles of Hierarchy

Extreme Programming in a Nutshell

Extreme Programming involves −

- **Writing unit tests before programming** and keeping all of the tests running at all times. The unit tests are automated and eliminates defects early, thus reducing the costs.
- Starting with a simple design just enough to code the features at hand and redesigning when required.
- Programming in pairs (called **pair programming**), with two programmers at one screen, taking turns to use the keyboard. While one of them is at the keyboard, the other constantly reviews and provides inputs.
- Integrating and testing the whole system several times a day.
- Putting a minimal working system into the production quickly and upgrading it whenever required.
- Keeping the customer involved all the time and obtaining constant feedback.

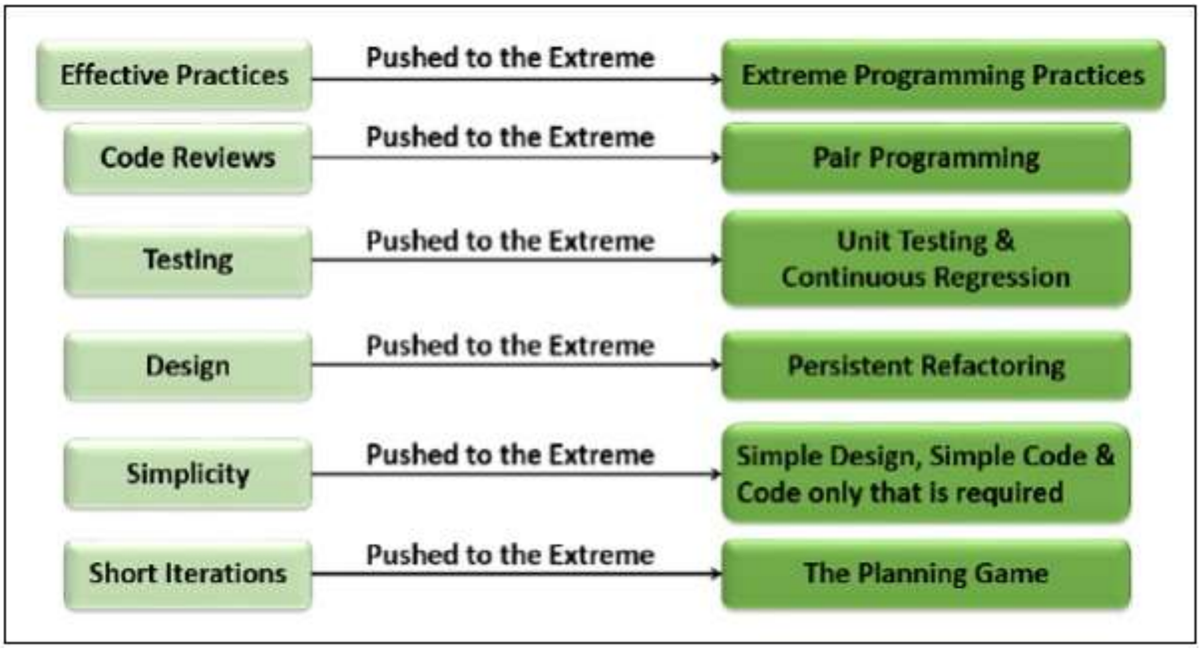
Iterating facilitates the accommodating changes as the software evolves with the changing requirements.



Why is it called “Extreme?”

Extreme Programming takes the effective principles and practices to extreme levels.

- Code reviews are effective as the code is reviewed all the time.
- Testing is effective as there is continuous regression and testing.
- Design is effective as everybody needs to do refactoring daily.
- Integration testing is important as integrate and test several times a day.
- Short iterations are effective as the planning game for release planning and iteration planning.



Java examples of the Law of Demeter

The Law of Demeter is often described this way:

> “Only talk to your immediate friends.”

or, put another way:

> “Don't talk to strangers.”

A “Law of Demeter” Java example

Before getting into the theory, I think the Law of Demeter is best explained through a source code example, in this case, a Java example. Here then is a Java class which attempts to demonstrate what method calls are considered "okay" according to the Law of Demeter:

```java
public class LawOfDemeterInJava
{
  private Topping cheeseTopping;

  /**
   * Good examples of following the Law of Demeter.
   */
  public void goodExamples(Pizza pizza)
  {
    Foo foo = new Foo();

    // (1) it's okay to call our own methods
    doSomething();

    // (2) it's okay to call methods on objects passed in to our method
    int price = pizza.getPrice();

    // (3) it's okay to call methods on any objects we create
    cheeseTopping = new CheeseTopping();
    float weight = cheeseTopping.getWeightUsed();

    // (4) any directly held component objects
    foo.doBar();
  }

  private void doSomething()
  {
    // do something here ...
  }
}
```

Now that you've seen the Law of Demeter in a Java example, here are the more formal rules for this law, as shown on the Wikipedia page:

The Law of Demeter for functions requires that a method M of an object O may only invoke the methods of the following kinds of objects:

1. O itself
2. M's parameters
3. Any objects created/instantiated within M
4. O's direct component objects
5. A global variable, accessible by O, in the scope of M

Open Closed principle

One word about **Open Closed principle** is a design principle which says that a class should be **open for extension but closed for modification.** Open Closed Principles is one of the principle from SOLID design principle where it represents "O".

In Simple language Open closed design principles says that **new functionality should be added by introducing new classes, methods or fields instead of modifying already tried and tested code. One of the way to achieve this is Inheritance** where class is extended to introduce new functionality on top of inherited basic features.

Benefit or Open Closed Design Principle:
1) Application will be more robust because we are not changing already tested class.
2) Flexible because we can easily accommodate new requirements.
3) Easy to test and less error prone.

The Liskov Substitution Principle (LSP): *functions that use pointers to base classes must be able to use objects of derived classes without knowing it*.

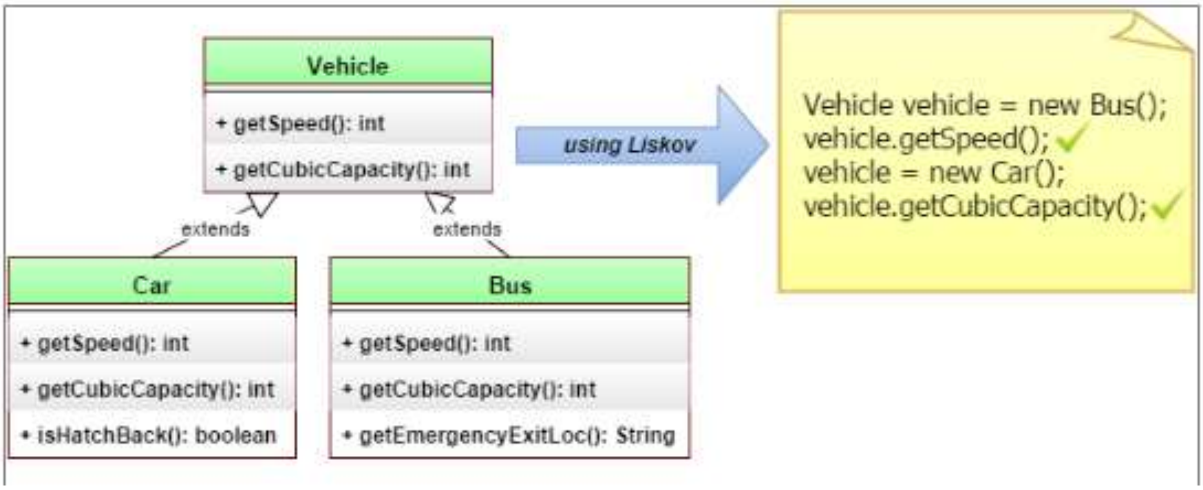
Liskov Substitution Principle explained

As the name suggests, Liskov Substitution Principle prescribes substitutability of a class by its subclass. Broadly put, a class can be replaced by its subclass in all practical usage scenarios.
This is actually in line with what Java also allows. A superclass reference can hold a subclass object i.e. superclass can be replaced by subclass in a superclass reference at any time. So, Java inheritance mechanism follows Liskov Substitution Principle.

Also note that, Liskov Substitution Principle is one of the five *SOLID Design Principles*. Specifically the L in SOLID stands for this principle.

Liskov Substitution Principle: Example in Java

Lets say we have a class Vehicle and its two sub-classes Car & Bus as shown below:



```
Vehicle vehicle = new Bus();
vehicle.getSpeed(); ✓
vehicle = new Car();
vehicle.getCubicCapacity(); ✓
```

Now, we can assign an object of type Car or that of type Bus to a reference of type Vehicle. All the functionality which is inherent in base class Vehicle, and is acquired by Bus and Car via inheritance, can be invoked on a reference of type Vehicle. Referring to the diagram above, it is possible to invoke methods like `getSpeed()` & `getCubicCapacity()` on a Vehicle reference which actually holds a Bus/Car object. The actual object's overridden implementation of these methods will be actually invoked. This is exactly what the Liskov Substitution Principle also states – subtype objects can replace super type objects without affecting the functionality inherent in the super type.

Liskov Violation Scenario Example: The Circle-Ellipse Problem

All circles are inherently ellipses with their major and minor axes being equal. In terms of classes if we make a class Ellipse then Circle class will be a child of Ellipse class i.e. it will extend Ellipse class. Nothing wrong in it so far.

Lets now use the Liskov Substitution Principle. In this case an object of type Circle can be assigned to a reference of type Ellipse. So, all the methods in Ellipse can/could be invoked on this object of Circle which is stored in it. One inherent functionality of an ellipse is that its stretchable. I.e. the length of the two axes of an ellipse can be changed. Lets say we have methods `setLengthOfAxisX()` and `setLengthOfAxisY()` through which the length of the two axes X & Y of an ellipse can altered.

However, calling any of these two methods on an object of type circle inside a reference of type ellipse would lead to a circle no longer being a circle as in a circle the length of the major and minor axes *have* to be equal. This is known as the *Circle-Ellipse Problem*.

This is a typical violation scenario of Liskov Substitution Principle as assigning a subtype object to a super type reference doesn't work which is not in line with the principle. The principle actually expects it to work smoothly.

Relation of Liskov Substitution Principle with Open Closed Principle
Open Closed Principle says that a class should be open for extension and closed for modification. I.e. to modify what a class does, we should not change the original class. Rather, we should override the original class and implement the functionality to be changed in the overriding class. This way when the derived class's(or the sub-type's) object is used in place of the parent/super-class, then the overridden functionality is executed on executing the same functionality. This is exactly in line with the Liskov Principle we just saw above.

Summary**: We looked at what is Liskov Substitution Principle and saw an example of it in Java. Next, we looked at the** *Circle-Ellipse Problem* **which is an example of violation of Liskov Substitution Principle. Lastly, we looked at how closely Open Closed Principle and Liskov Substitution Principle are related.**

Dependency Inversion Principle

As a Java programmer, you've likely heard about code coupling and have been told to avoid tightly coupled code. Ignorance of writing "*good code*" is the main reason of tightly coupled code existing in applications. As an example, creating an object of a class using the **new** operator results in a class being tightly coupled to another class. Such coupling appears harmless and does not disrupt small programs. But, as you move into enterprise application development, tightly coupled code can lead to serious adverse consequences.

When one class knows explicitly about the design and implementation of another class, changes to one class raise the risk of breaking the other class. Such changes can have rippling effects across the application making the application fragile. To avoid such problems, you should write "*good code*" that is loosely coupled, and to support this you can turn to the Dependency Inversion Principle.

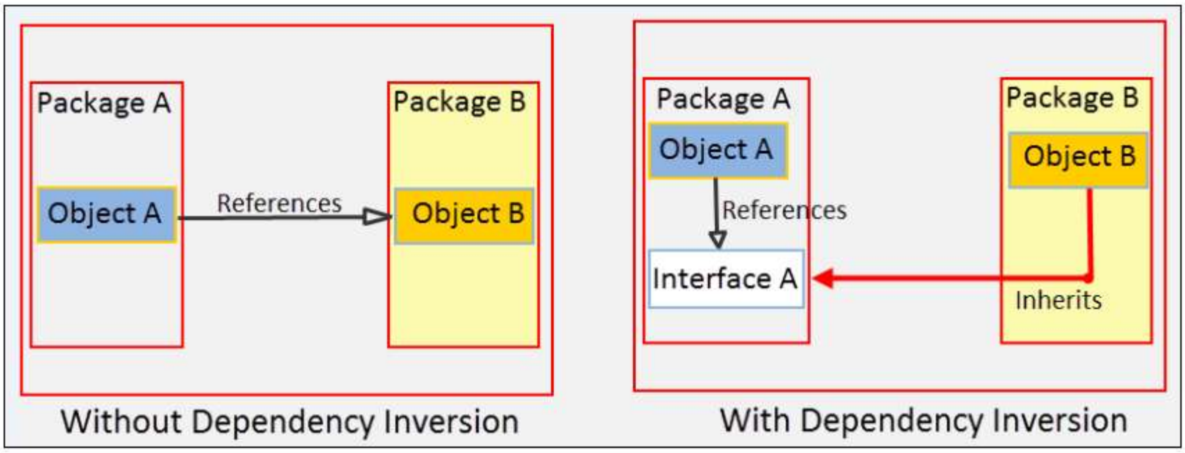
The Dependency Inversion Principle represents the last "D" of the five SOLID principles of object-oriented programming. Robert C. Martin first postulated the Dependency Inversion Principle and published it in 1996. The principle states:

> *"A. High-level modules should not depend on low-level modules. Both should depend on abstractions.*

B. Abstractions should not depend on details. Details should depend on abstractions."

Conventional application architecture follows a top-down design approach where a high-level problem is broken into smaller parts. In other words, the high-level design is described in terms of these smaller parts. As a result, high-level modules that gets written directly depends on the smaller (low-level) modules.

What Dependency Inversion Principle says is that instead of a high-level module depending on a low-level module, both should depend on an abstraction. Let us look at it in the context of Java through this figure.



In the figure above, without Dependency Inversion Principle, Object A in Package A refers Object B in Package B. With Dependency Inversion Principle, an Interface A is introduced as an abstraction in Package A. Object A now refers Interface A and Object B inherits from Interface A. What the principle has done is:

1. **Both Object A and Object B now depends on Interface A, the abstraction.**
2. **It inverted the dependency that existed from Object A to Object B into Object B being dependent on the abstraction (Interface A).**

Before we write code that follows the Dependency Inversion Principle, let's examine a typical violating of the principle.

Design Pattern

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

What is Gang of Four (GOF)?

In 1994, four authors 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. These authors are collectively known as **Gang of Four (GOF)**.

Usage of Design Pattern

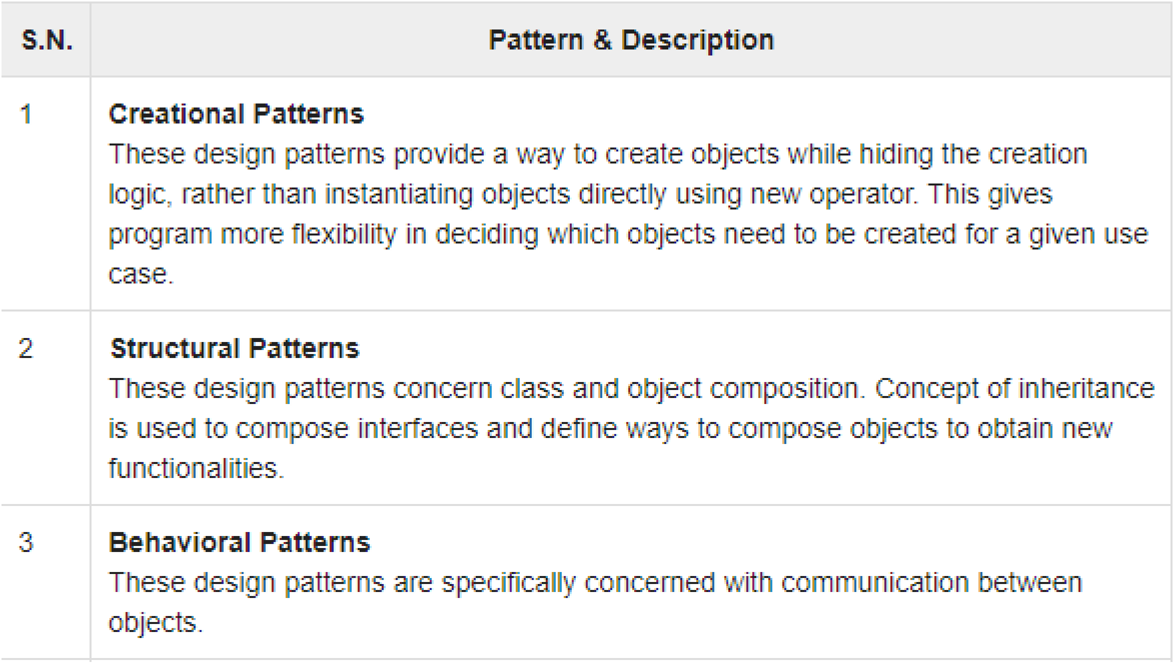
Design Patterns have two main usages in software development.

Common platform for developers

Design patterns provide a standard terminology and are specific to particular scenario. For example, a singleton design pattern signifies use of single object so all developers familiar with single design pattern will make use of single object and they can tell each other that program is following a singleton pattern.

Best Practices

Design patterns have been evolved over a long period of time and they provide best solutions to certain problems faced during software development. Learning these patterns helps inexperienced developers to learn software design in an easy and faster way.



S.N.	Pattern & Description
1	**Creational Patterns** These design patterns provide a way to create objects while hiding the creation logic, rather than instantiating objects directly using new operator. This gives program more flexibility in deciding which objects need to be created for a given use case.
2	**Structural Patterns** These design patterns concern class and object composition. Concept of inheritance is used to compose interfaces and define ways to compose objects to obtain new functionalities.
3	**Behavioral Patterns** These design patterns are specifically concerned with communication between objects.

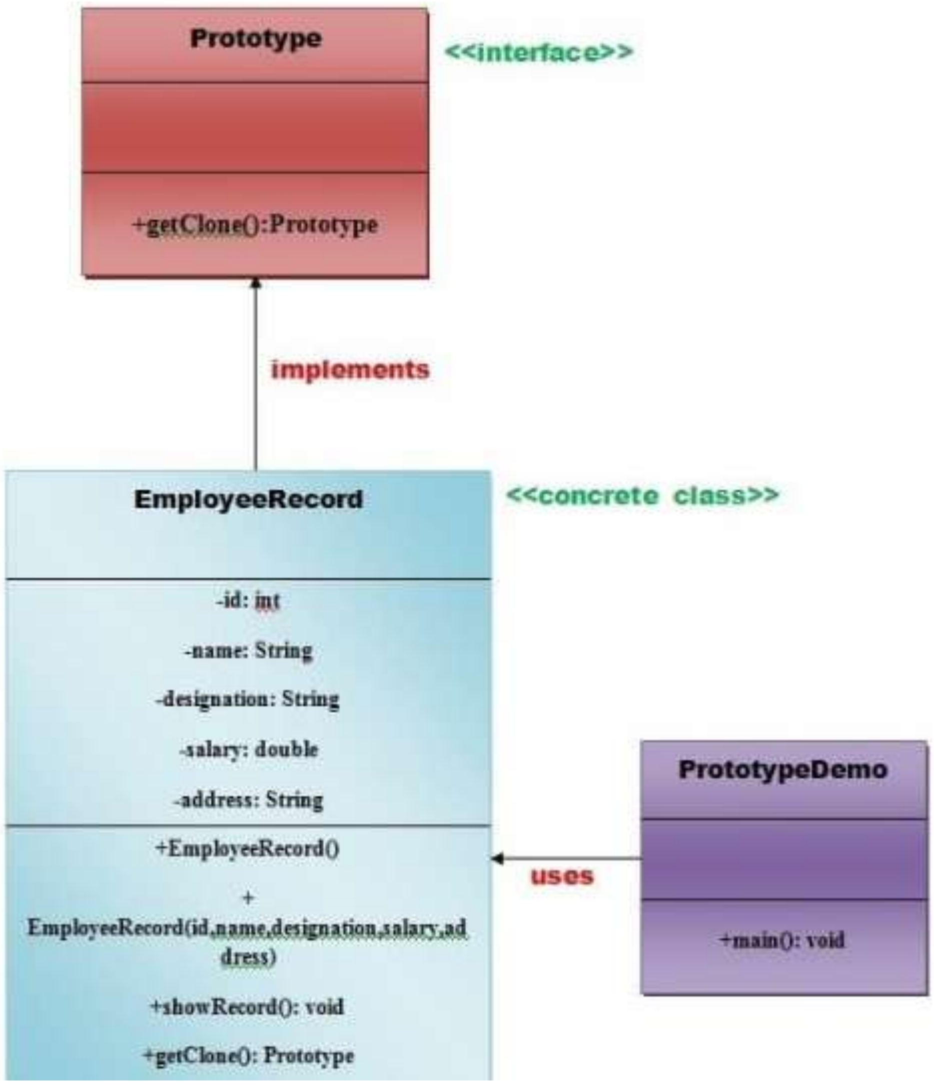
Creational design patterns

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.

Prototype Design Pattern

Prototype Pattern says that **cloning of an existing object instead of creating new one and can also be customized as per the requirement**.

This pattern should be followed, if the cost of creating a new object is expensive and resource intensive.



Example of Prototype Design Pattern

Let's see the example of prototype design pattern.

File: Prototype.java

```
interface Prototype {

    public Prototype getClone();

}//End of Prototype interface.
```

File: EmployeeRecord.java

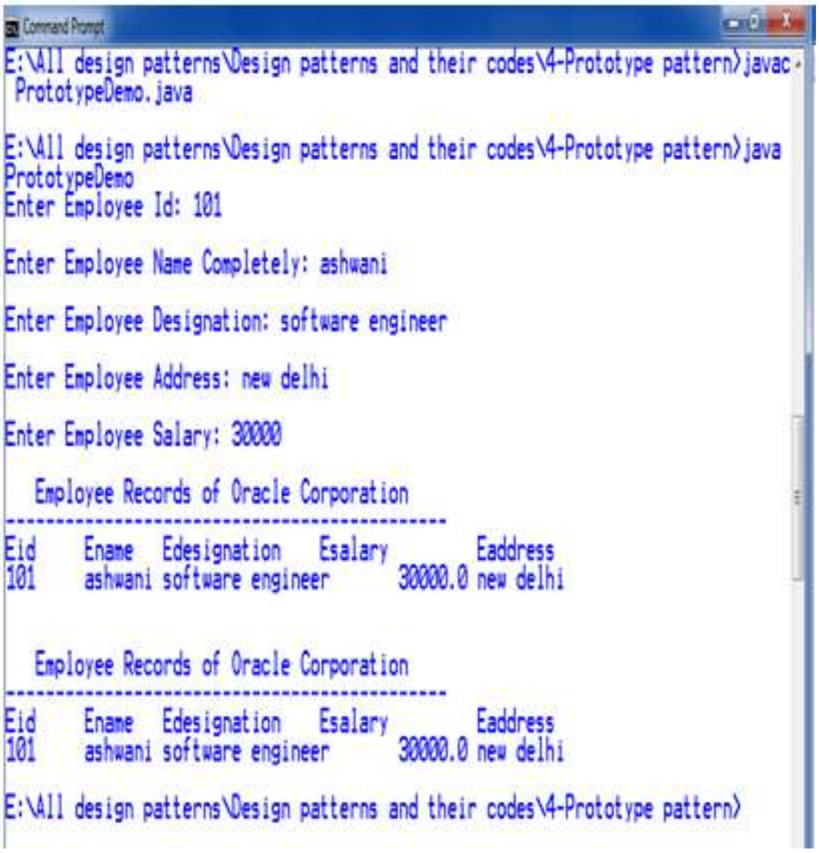
```
class EmployeeRecord implements Prototype{

  private int id;
  private String name, designation;
```

```
   private double salary;
   private String address;

   public EmployeeRecord(){
            System.out.println("   Employee Records of Oracle Corporation ");
            System.out.println("---------------------------------------------");
            System.out.println("Eid"+"\t"+"Ename"+"\t"+"Edesignation"+"\t"+"Esalary"+"\t\t"+"Eaddress");

}

 public  EmployeeRecord(int id, String name, String designation, double salary, String address) {

        this();
        this.id = id;
        this.name = name;
        this.designation = designation;
        this.salary = salary;
        this.address = address;
    }

  public void showRecord(){

        System.out.println(id+"\t"+name+"\t"+designation+"\t"+salary+"\t"+address);
   }

    @Override
    public Prototype getClone() {

        return new EmployeeRecord(id,name,designation,salary,address);
    }
}//End of EmployeeRecord class.
```

File: PrototypeDemo.java

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

class PrototypeDemo{
     public static void main(String[] args) throws IOException {

        BufferedReader br =new BufferedReader(new InputStreamReader(System.in));
        System.out.print("Enter Employee Id: ");
        int eid=Integer.parseInt(br.readLine());
        System.out.print("\n");

        System.out.print("Enter Employee Name: ");
        String ename=br.readLine();
        System.out.print("\n");

        System.out.print("Enter Employee Designation: ");
        String edesignation=br.readLine();
        System.out.print("\n");

        System.out.print("Enter Employee Address: ");
        String eaddress=br.readLine();
        System.out.print("\n");

        System.out.print("Enter Employee Salary: ");
        double esalary= Double.parseDouble(br.readLine());
        System.out.print("\n");

        EmployeeRecord e1=new EmployeeRecord(eid,ename,edesignation,esalary,eaddress);

        e1.showRecord();
        System.out.println("\n");
        EmployeeRecord e2=(EmployeeRecord) e1.getClone();
        e2.showRecord();
    }
}//End of the ProtoypeDemo class.
```

Output



```
E:\All design patterns\Design patterns and their codes\4-Prototype pattern>javac
 PrototypeDemo.java

E:\All design patterns\Design patterns and their codes\4-Prototype pattern>java
PrototypeDemo
Enter Employee Id: 101

Enter Employee Name Completely: ashwani

Enter Employee Designation: software engineer

Enter Employee Address: new delhi

Enter Employee Salary: 30000

   Employee Records of Oracle Corporation
---------------------------------------------
Eid     Ename   Edesignation    Esalary         Eaddress
101     ashwani software engineer       30000.0 new delhi


   Employee Records of Oracle Corporation
---------------------------------------------
Eid     Ename   Edesignation    Esalary         Eaddress
101     ashwani software engineer       30000.0 new delhi

E:\All design patterns\Design patterns and their codes\4-Prototype pattern>
```

Factory Method

Factory method is a creational design pattern, i.e., **related to object creation.** In Factory pattern, we create object without exposing the creation logic to client and the client use the same common interface to create new type of object.
The idea is to use a static member-function (static factory method) which creates & returns instances, hiding the details of class modules from user.

Other examples of Factory Method:

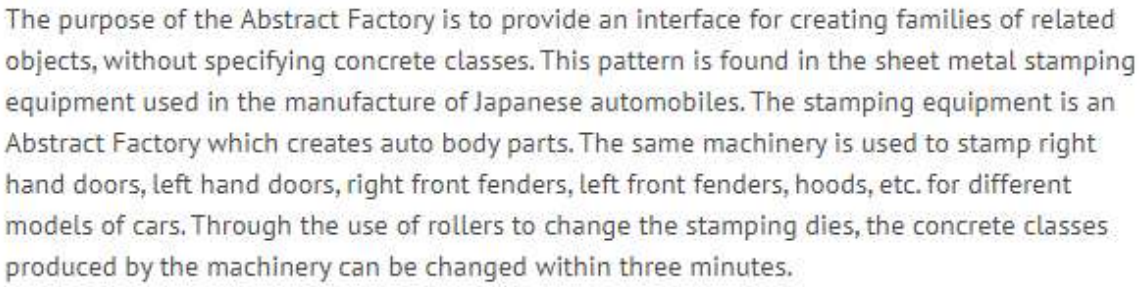
In travel site, we can book train ticket as well bus tickets and flight ticket. In this case user can give his **travel type as 'bus', 'train' or 'flight'**.

Here we have an **abstract class 'AnyTravel' with a static member function 'GetObject' which depending on user's travel type, will create & return object of 'BusTravel' or ' TrainTravel'.**

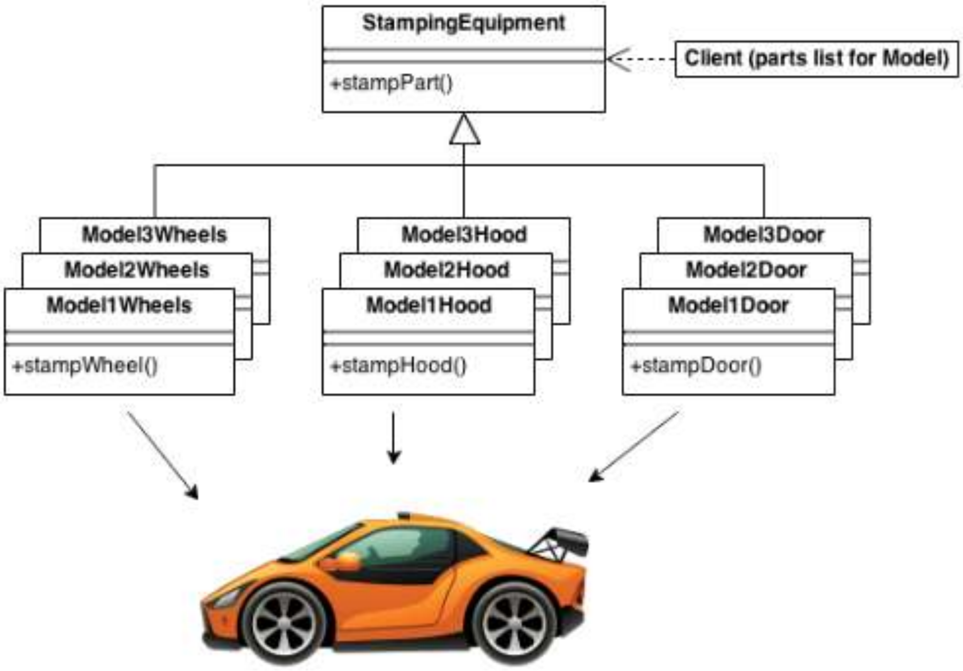
'BusTravel' or ' TrainTravel' have common functions like passenger name, Origin, destinationparameters.

Abstract Factory patterns work around a super-factory which creates other factories. This factory is also called as factory of factories. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.

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



The purpose of the Abstract Factory is to provide an interface for creating families of related objects, without specifying concrete classes. This pattern is found in the sheet metal stamping equipment used in the manufacture of Japanese automobiles. The stamping equipment is an Abstract Factory which creates auto body parts. The same machinery is used to stamp right hand doors, left hand doors, right front fenders, left front fenders, hoods, etc. for different models of cars. Through the use of rollers to change the stamping dies, the concrete classes produced by the machinery can be changed within three minutes.



Both Abstract Factory and Factory design pattern are creational design pattern and use to decouple clients from creating object they need, But there is a significant difference between Factory and Abstract Factory design pattern, Factory design pattern produces implementation of `Products` e.g. Garment Factory produce different kinds of clothes, On the other hand Abstract Factory design pattern adds another layer of abstraction over Factory Pattern and Abstract Factory implementation itself e.g. Abstract Factory will allow you to choose a particular Factory implementation based upon need which will then produce different kinds of products.

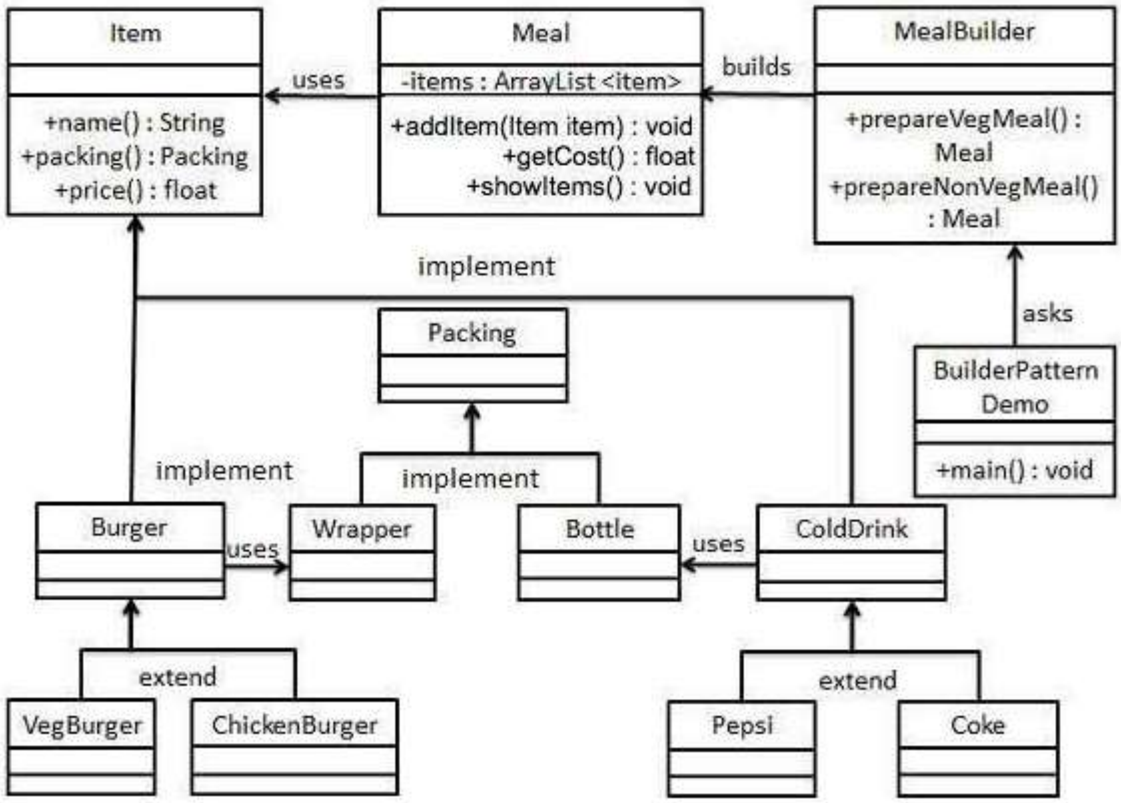
In short

1) `Abstract Factory` design pattern creates `Factory`

2) `Factory design pattern` creates `Products`

Design Patterns - Builder Pattern

Builder pattern builds a complex object using simple objects and using a step by step approach. A Builder class builds the final object step by step. This builder is independent of other objects.



Design Pattern - Singleton Pattern

The singleton pattern is one of the simplest design patterns. Sometimes we need **to have only one instance of our class for example a single DB connection shared by multiple objects as creating a separate DB connection for every object may be costly.** Similarly, there can be a single configuration manager or error manager in an application that handles all problems instead of creating multiple managers.

Definition:

The singleton pattern is a design pattern that restricts the instantiation of a class to one 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(){
      System.out.println("Hello World!");
   }
}
```

Step 2

Get the only object from the singleton class.

SingletonPatternDemo.java

```
public class SingletonPatternDemo {
   public static void main(String[] args) {

      //illegal construct
      //Compile Time Error: The constructor SingleObject() is not
visible
      //SingleObject object = new SingleObject();

      //Get the only object available
      SingleObject object = SingleObject.getInstance();

      //show the message
      object.showMessage();
   }
}
```

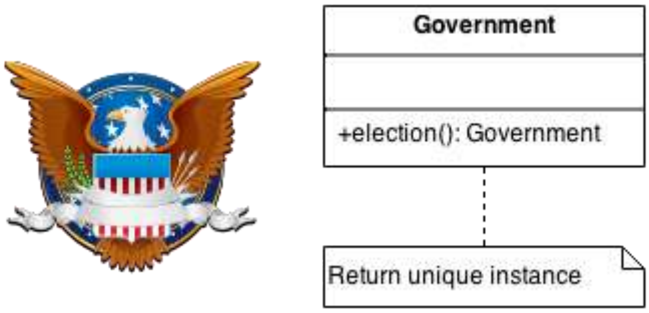
Step 3

Verify the output.

```
Hello World!
```

Example

The Singleton pattern ensures that a class has only one instance and provides a global point of access to that instance. It is named after the singleton set, which is defined to be a set containing one element. **The office of the President of the United States is a Singleton. The United States Constitution specifies the means by which a president is elected, limits the term of office, and defines the order of succession. As a result, there can be at most one active president at any given time. Regardless of the personal identity of the active president, the title, "The President of the United States" is a global point of access that identifies the person in the office.**



What is Dependency Injection Design Pattern?

The dependency injection design pattern is a way of object configuration to obtain independency of each object responsibilities. In there, a dependent object is configured from the outside instead of configuring inside the same object. It enables loose coupling of applications with a set of general guidelines and it is not a library, framework or tool. This also popular as Hollywood principle – "Don't call us – we'll call you".

Instead of hard-coding dependencies, such as specifying a database server, list of services are injected into the component via a third party. So that component needs not to worry about creating and handling those external services. It leads to many benefits especially when the system gets larger and complex.

What is Injection?

Injecting is inserting or passing the data or service to the dependent entity. A dependency can be pushed into the class from the outside.

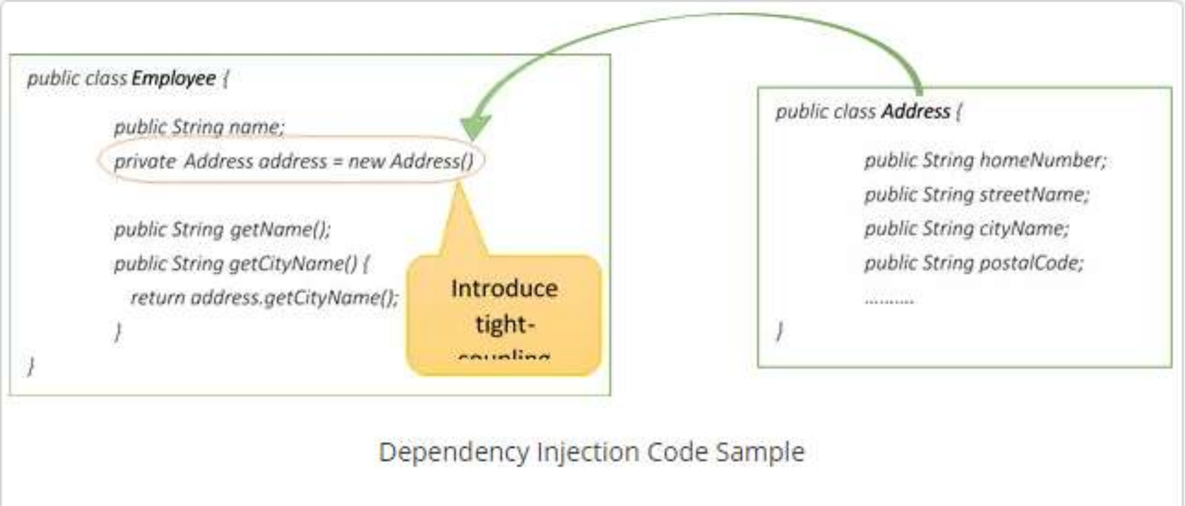
Simple Real Life Example

Can you live without using a toothbrush? It is a – can't live without, item on your personal care item list. But, have you ever thought how and where your toothbrush was made? Most probably, 'No' or 'Who cares?' will be your answers. Of course, it is totally out of our scope. Making a standard toothbrush is a complex, time-consuming process. It needs a list of production materials and labor aligned with a standard process before coming to our hand. So, no way we can build it at our home. The toothbrush manufacturer does it for us. We just have to buy it from the shop and use it. In this scenario, you depend on the toothbrush to clean your teeth. Since, you can't make the toothbrush by your own, the manufacturer makes that for you that is **manufacturer inject the toothbrush dependency on yourself so that you can use it to keep you clean.** Likewise, dependency injection is a way of providing an external data or service without interrupting individual existence of an entity.

Without Dependency Injection

Let's assume that small business wants to keep track of their employees. It creates the 'Employee' class and stores the relevant data like name, id, address and phone number. Since the address contains several parts and it may vary depending on certain conditions designers decide to make it as a separate class. Now, when you are creating or using an employee object, an address object should be instantiated within the employee object in order to complete the proper behavior of employee object.

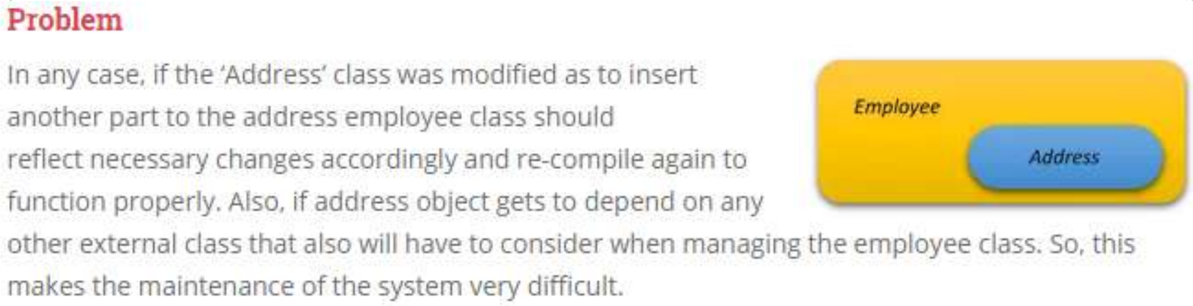
Let's look at the code samples,



```
public class Employee {

        public String name;
        private Address address = new Address()

        public String getName();
        public String getCityName() {
          return address.getCityName();
        }
}
```

```
public class Address {

        public String homeNumber;
        public String streetName;
        public String cityName;
        public String postalCode;
        ..........
}
```

Dependency Injection Code Sample

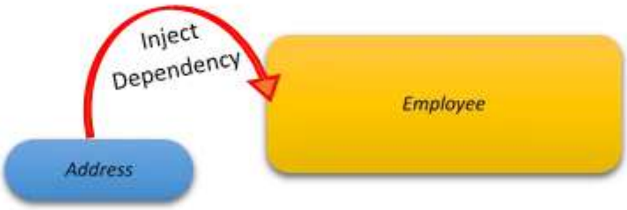


Problem

In any case, if the 'Address' class was modified as to insert another part to the address employee class should reflect necessary changes accordingly and re-compile again to function properly. Also, if address object gets to depend on any other external class that also will have to consider when managing the employee class. So, this makes the maintenance of the system very difficult.

Solution

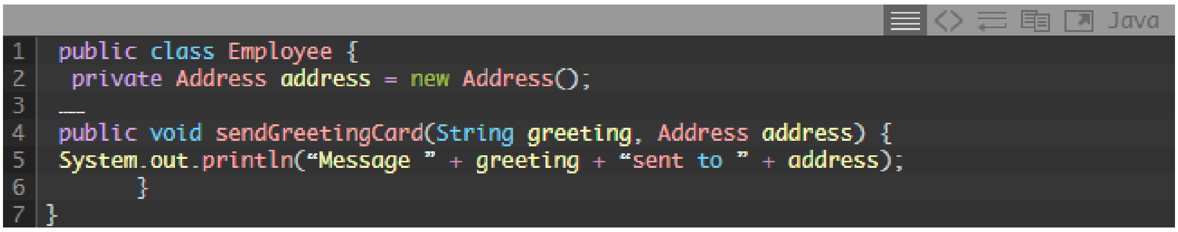
Find a way to decouple the Address object from the Employee object. Separating each class responsibilities independently, by introducing a way to inject the service from Address object to Employee object. So that Employee object can behave independently.



Methods of Injecting Dependency

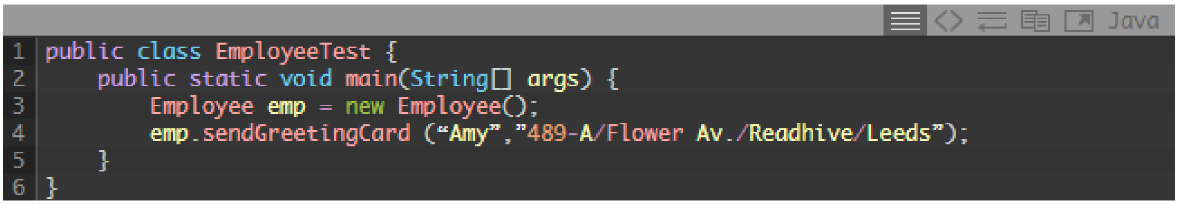
There are 3 main methods in use when implementing a less coupled code with the help of dependency injection design pattern. Developer selects a suitable approach depending on conditions at the hand.

1. Constructor Injection – Dependencies are inserted via class constructor
2. Setter Injection – Client or dependency provider provides a setter method which is used by the injector()
3. Interface Injection – The dependency provider introduces an injector method that will inject the dependency into the client



```java
 public class Employee {
  private Address address = new Address();
 ……
 public void sendGreetingCard(String greeting, Address address) {
 System.out.println(“Message ” + greeting + “sent to ” + address);
       }
}
```

Testing class contains below code,



```java
public class EmployeeTest {
    public static void main(String[] args) {
        Employee emp = new Employee();
        emp.sendGreetingCard (“Amy”,”489-A/Flower Av./Readhive/Leeds”);
    }
}
```

Setter Injection

In this approach, setter method is used to pass the externally instantiated object to the required class. Also, it is encouraging to apply optional dependencies in setter injection. The dependent class should be able to function even without the setter arguments. These dependencies can be changed even after the object is instantiated and it won't create a new instance always like with constructor injection. Hence, setter injection is flexible than constructor injection.

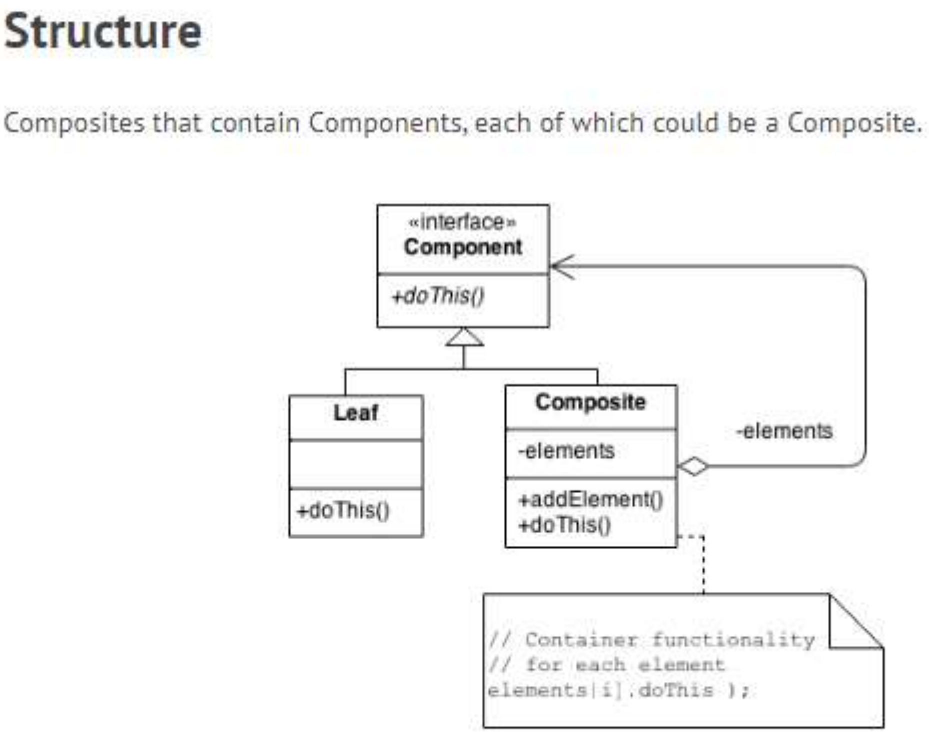
```java
public class Employee {
 private Address address = null;

 public void setAddress(Address address){
 this.address = address;
 }
        public Address getAddress() {
           return this.address;
        }
 public void sendGreetingCard(String name, Address address) {
 System.out.println(“Message ” + greeting + “sent to ” + address);
        }
}
```

Structural patterns

Composite Design Pattern

Intent

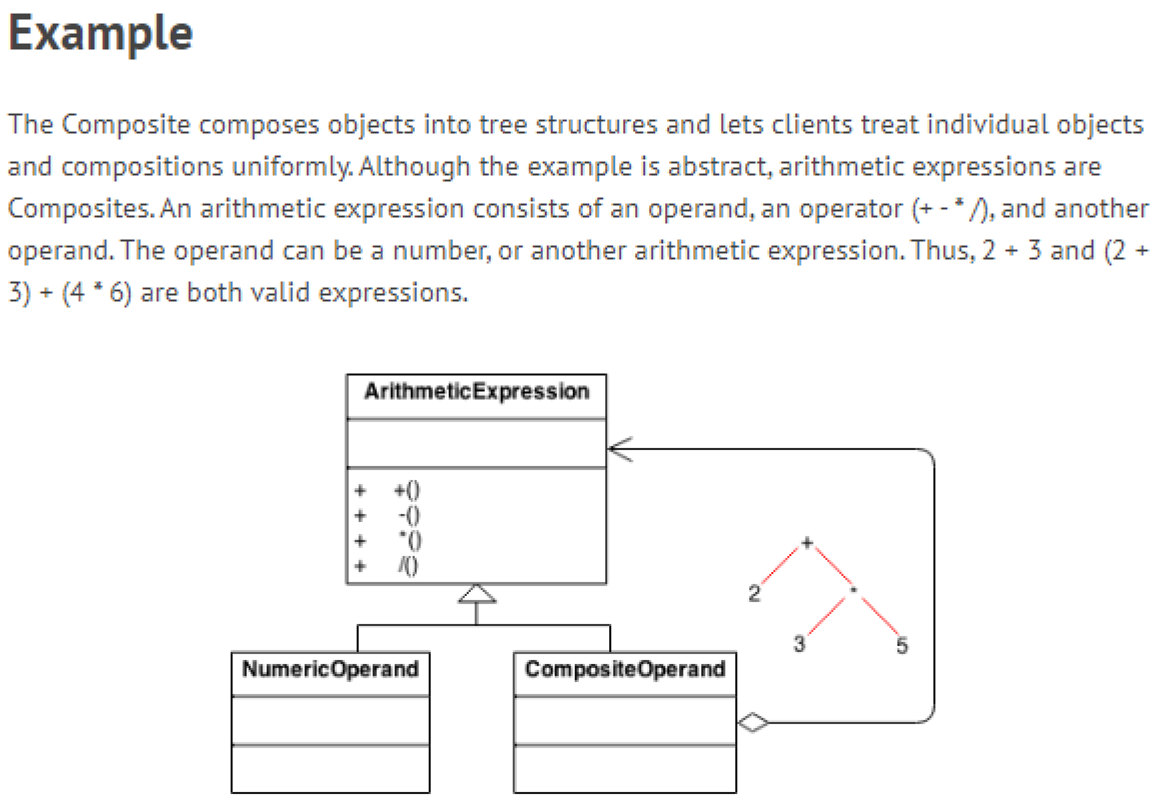
- Compose objects into tree structures to represent whole-part hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
- Recursive composition
- "Directories contain entries, each of which could be a directory."
- 1-to-many "has a" up the "is a" hierarchy



Structure

Composites that contain Components, each of which could be a Composite.

```
// Container functionality
// for each element
elements[i].doThis );
```



Example

The Composite composes objects into tree structures and lets clients treat individual objects and compositions uniformly. Although the example is abstract, arithmetic expressions are Composites. An arithmetic expression consists of an operand, an operator (+ - * /), and another operand. The operand can be a number, or another arithmetic expression. Thus, 2 + 3 and (2 + 3) + (4 * 6) are both valid expressions.

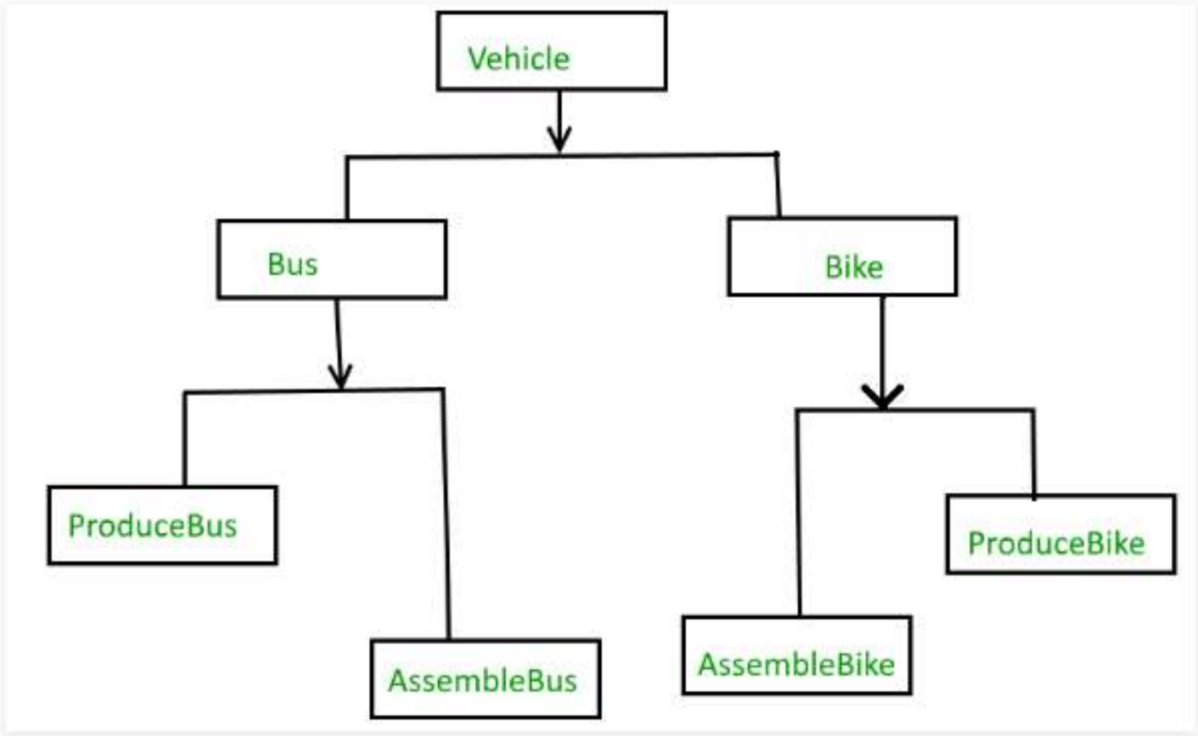
Bridge Design Pattern

The Bridge design pattern allows you to **separate the abstraction from the implementation.** It is a structural design pattern.

There are 2 parts in Bridge design pattern :

1. Abstraction
2. Implementation

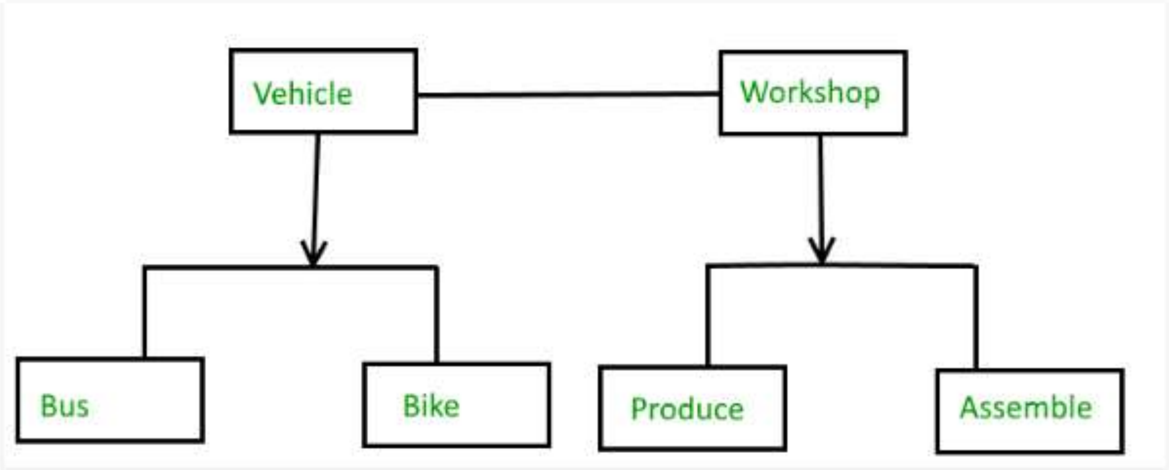
Without Bridge Design Pattern



But the above solution has a problem. If you want to change the Bus class, then you may end up changing ProduceBus and AssembleBus as well and if the change is workshop specific then you may need to change the Bike class as well.

With Bridge Design Pattern

You can solve the above problem by decoupling the Vehicle and Workshop interfaces in the below manner.



Proxy Design Pattern

Proxy means ‘in place of’, representing’ or ‘in place of’ or ‘on behalf of’ are literal meanings of proxy and that directly explains **Proxy Design Pattern**.
Proxies are also called surrogates, handles, and wrappers.

A real world example can be a cheque or credit card is a proxy for what is in our bank account. It can be used in place of cash, and provides a means of accessing that cash when required. And that’s exactly what the Proxy pattern does – “**Controls and manage access to the object they are protecting**“.

Flyweight Design Pattern

As per GoF definition, **flyweight design pattern** enables use sharing of objects to support large numbers of fine-grained objects efficiently. A flyweight is a **shared object** that can be used in multiple contexts simultaneously. The flyweight acts as an independent object in each context.

1. When to use flyweight design pattern

We can use flyweight pattern in following scenarios:

- When we need a large number of similar objects that are unique in terms of only a few parameters and most of the stuffs are common in general.
- We need to control the memory consumption by large number of objects – by creating fewer objects and sharing them across.

2. Extrinsic and intrinsic attributes

A flyweight objects essentially has two kind of attributes – intrinsic and extrinsic.

An **intrinsic** state attribute is stored/shared in the flyweight object, and it is independent of flyweight’s context. As the best practice, we should make intrinsic states **immutable**.

An **extrinsic** state varies with flyweight’s context, which is why they cannot be shared. Client objects maintain the extrinsic state, and they need to pass this to a flyweight object during object creation.

3. Real world example of flyweight pattern

- Suppose we have a **pen** which can exist with/without **refill**. A refill can be of any color thus a pen can be used to create drawings having N number of colors.

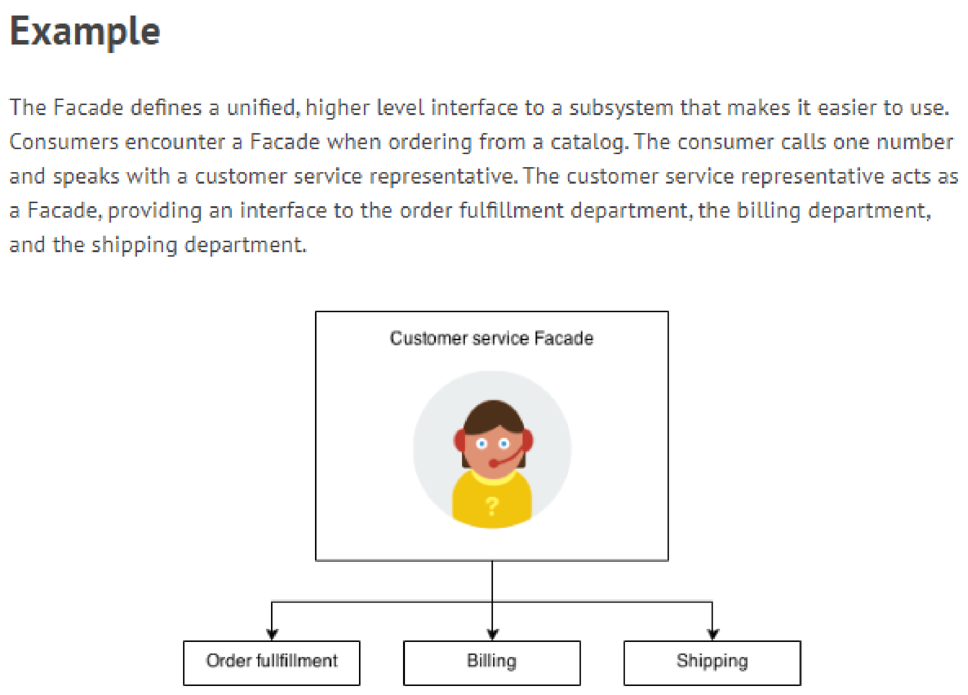
  Here `Pen` can be flyweight object with `refill` as extrinsic attribute. All other attributes such as pen body, pointer etc. can be intrinsic attributes which will be common to all pens. A pen will be distinguished by its refill color only, nothing else.

  All application modules which need to access a red pen – can use the same instance of red pen (shared object). Only when a different color pen is needed, application module will ask for another pen from flyweight factory.

Facade Design Pattern

Intent

- Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
- Wrap a complicated subsystem with a simpler interface.



Example

The Facade defines a unified, higher level interface to a subsystem that makes it easier to use. Consumers encounter a Facade when ordering from a catalog. The consumer calls one number and speaks with a customer service representative. The customer service representative acts as a Facade, providing an interface to the order fulfillment department, the billing department, and the shipping department.

Decorator Design Pattern

Intent

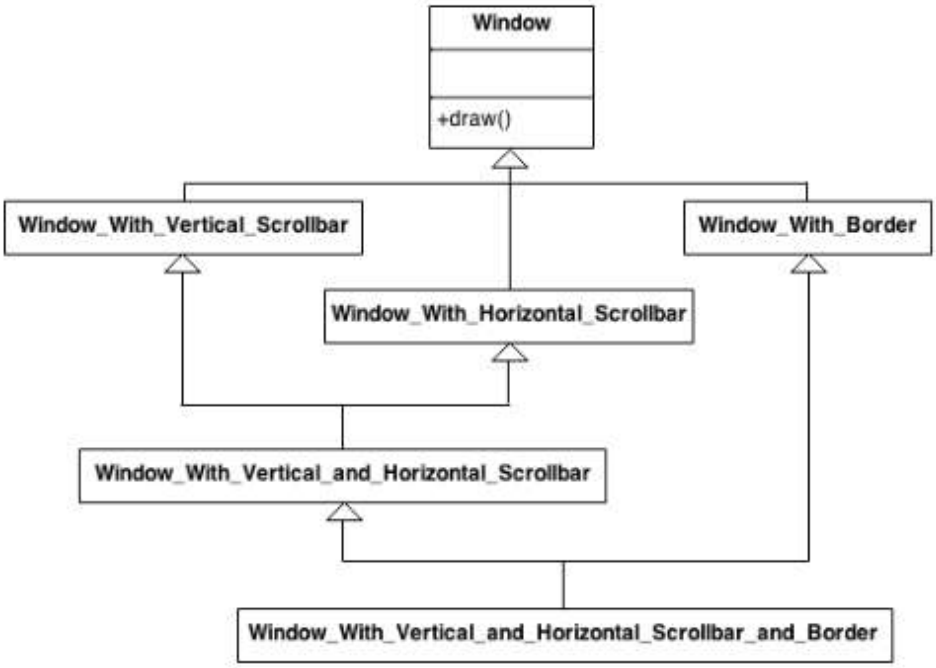
- Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
- Client-specified embellishment of a core object by recursively wrapping it.
- Wrapping a gift, putting it in a box, and wrapping the box.

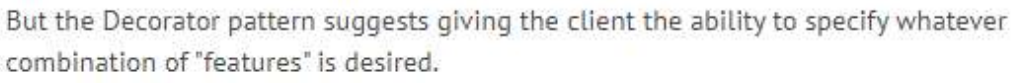
Problem

You want to add behavior or state to individual objects at run-time**. Inheritance is not feasible** because it is static and applies to an entire class.

Discussion

Suppose you are working on a user interface toolkit and you wish to support adding borders and scroll bars to windows. You could define an **inheritance hierarchy** like ...


Window
+draw()
Window_With_Vertical_Scrollbar
Window_With_Border
Window_With_Horizontal_Scrollbar
Window_With_Vertical_and_Horizontal_Scrollbar
Window_With_Vertical_and_Horizontal_Scrollbar_and_Border


But the Decorator pattern suggests giving the client the ability to specify whatever combination of "features" is desired.

Module pattern

In software engineering, the **module pattern** is a design pattern used to implement the concept of software modules, defined by modular programming, in a programming language with incomplete direct support for the concept.

In Python, the pattern is built into the language, and each .py file is automatically a module.

The Module Pattern

The Module pattern was originally defined as a way to provide both private and public encapsulation for classes in conventional software engineering.

In JavaScript, the Module pattern is used to further *emulate* the concept of classes in such a way that we're able to include both public/private methods and variables inside a single object, thus shielding particular parts from the global scope. What this results in is a reduction in the likelihood of our function names conflicting with other functions defined in additional scripts on the page (Figure 9-2).

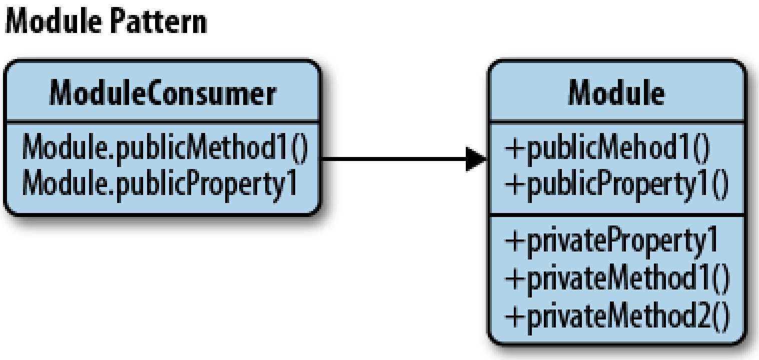


Figure 9-2. Module pattern

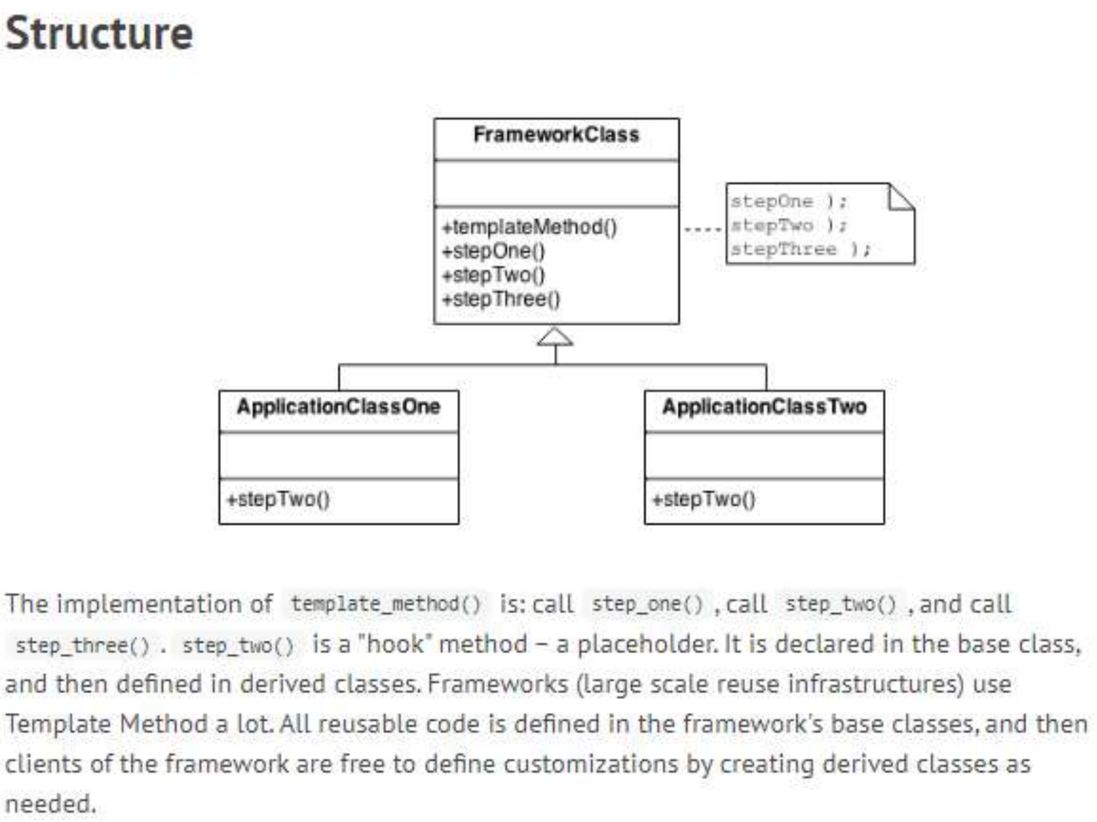
Behavioral patterns

Template Method Design Pattern

Defer the exact steps of an algorithm to a subclass

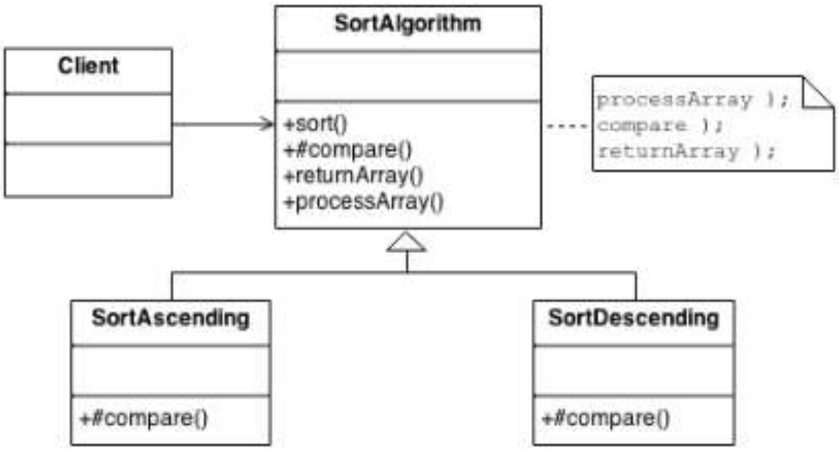
Intent

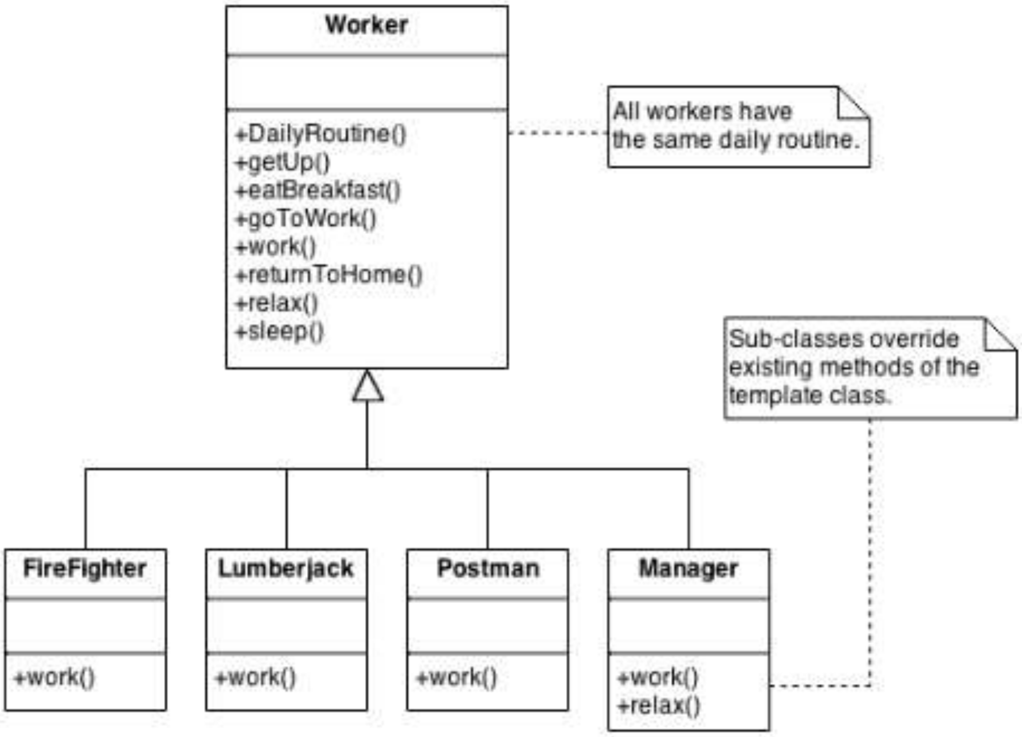
- Define the skeleton of an algorithm in an operation, deferring some steps to client subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
- Base class declares algorithm 'placeholders', and derived classes implement the placeholders.



Structure

The implementation of `template_method()` is: call `step_one()` , call `step_two()` , and call `step_three()` . `step_two()` is a "hook" method – a placeholder. It is declared in the base class, and then defined in derived classes. Frameworks (large scale reuse infrastructures) use Template Method a lot. All reusable code is defined in the framework's base classes, and then clients of the framework are free to define customizations by creating derived classes as needed.





Interpreter Design Pattern

A way to include language elements in a program

The best example of interpreter design pattern is java compiler that interprets the java source code into byte code that is understandable by JVM. Google Translator is also an example of interpreter pattern where the input can be in any language and we can get the output interpreted in another language.

This pattern is used in SQL parsing, symbol processing engine etc.

Concurrency design patterns

In software engineering, a design pattern is a solution to a common problem. This solution has been used many times, and it has proved to be an optimal solution to the problem. You can use them to avoid 'reinventing the wheel' every time you have to solve one of these problems. Singleton or Factory are the examples of common design patterns used in almost every application.

Concurrency also has its own design patterns. In this section, we describe some of the most useful concurrency design patterns and their implementation in the Java language.

Read-write lock

When you protect access to a shared variable with a lock, only one task can access that variable, independently of the operation you are going to perform on it. Sometimes, you will have variables that you modify a few times but read many times. In this circumstance, a lock provides poor performance because all the read operations can be made concurrently without any problem. To solve this problem, there exists the read-write lock design pattern. This pattern defines a special kind of lock with two internal locks: one for read operations and the other for write operations. The behavior of this lock is as follows:

> - If one task is doing a read operation and another task wants to do another read operation, it can do it
> - If one task is doing a read operation and another task wants to do a write operation, it's blocked until all the readers finish
> - If one task is doing a write operation and another task wants to do an operation (read or write), it's blocked until the writer finishes

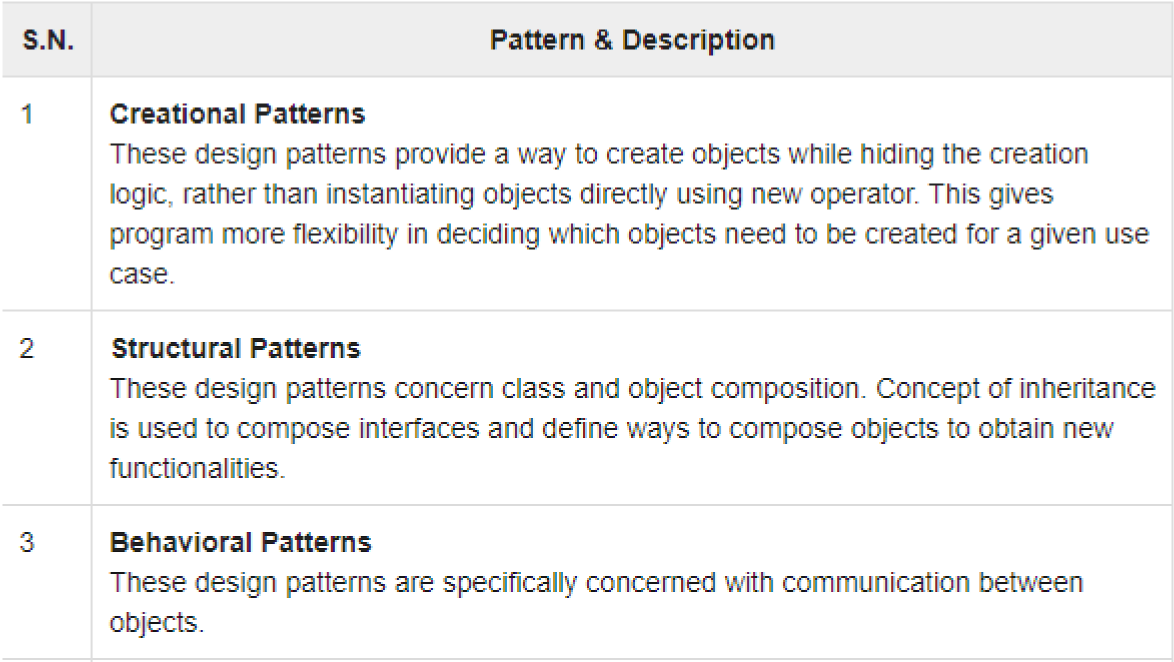
The Java concurrency API includes the class `ReentrantReadWriteLock` that implements this design pattern. If you want to implement this pattern from scratch, you have to be very careful with the priority between read-tasks and write-tasks. If too many read-tasks exist, write-tasks can be waiting too long.

SUMMARY

https://sourcemaking.com/design_patterns

https://www.tutorialspoint.com/design_pattern/index.htm

https://subscription.packtpub.com/book/application_development/9781785886126/1/ch01lvl1sec12/concurrency-design-patterns



S.N.	Pattern & Description
1	**Creational Patterns** These design patterns provide a way to create objects while hiding the creation logic, rather than instantiating objects directly using new operator. This gives program more flexibility in deciding which objects need to be created for a given use case.
2	**Structural Patterns** These design patterns concern class and object composition. Concept of inheritance is used to compose interfaces and define ways to compose objects to obtain new functionalities.
3	**Behavioral Patterns** These design patterns are specifically concerned with communication between objects.

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**
  Creates an instance of several families of classes
- **Builder**
  Separates object construction from its representation
- **Factory Method**
  Creates an instance of several derived classes
- **Object Pool**
  Avoid expensive acquisition and release of resources by recycling objects that are no longer in use

- **Prototype**
  A fully initialized instance to be copied or cloned
- **Singleton**
  A class of which only a single instance can exist

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**
  Match interfaces of different classes
- **Bridge**
  Separates an object’s interface from its implementation
- **Composite**
  A tree structure of simple and composite objects
- **Decorator**
  Add responsibilities to objects dynamically
- **Facade**
  A single class that represents an entire subsystem
- **Flyweight**
  A fine-grained instance used for efficient sharing
- **Private Class Data**
  Restricts accessor/mutator access
- **Proxy**
  An object representing another object

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**
  A way of passing a request between a chain of objects
- **Command**
  Encapsulate a command request as an object
- **Interpreter**
  A way to include language elements in a program
- **Iterator**
  Sequentially access the elements of a collection
- **Mediator**
  Defines simplified communication between classes
- **Memento**
  Capture and restore an object's internal state
- **Null Object**
  Designed to act as a default value of an object
- **Observer**
  A way of notifying change to a number of classes
- **State**
  Alter an object's behavior when its state changes

- **Strategy**
  Encapsulates an algorithm inside a class
- **Template method**
  Defer the exact steps of an algorithm to a subclass
- **Visitor**
  Defines a new operation to a class without change