# *Design Pattern*

[*Design Pattern* 1](#_Toc524423781)

[Design Pattern: 4](#_Toc524423782)

[What is Design Pattern? 4](#_Toc524423783)

[Introduction of GOF 5](#_Toc524423784)

[What is Gang of Four (GOF)? 5](#_Toc524423785)

[Types of Design Patterns 5](#_Toc524423786)

[Creational Design Pattern 6](#_Toc524423787)

[What is Creational Design Pattern? 6](#_Toc524423788)

[Structural Design Pattern 6](#_Toc524423789)

[What is Structural Design Pattern? 6](#_Toc524423790)

[Behavioral Design Pattern 7](#_Toc524423791)

[What is Behavioral Design Pattern? 7](#_Toc524423792)

[Abstract Factory 11](#_Toc524423793)

[Builder 11](#_Toc524423794)

[Factory Method /Factory 12](#_Toc524423795)

[Prototype 16](#_Toc524423796)

[Singleton 19](#_Toc524423797)

[Adapter 20](#_Toc524423798)

[Bridge 20](#_Toc524423799)

[Composite 20](#_Toc524423800)

[Decorator 20](#_Toc524423801)

[Facade 20](#_Toc524423802)

[Flyweight 20](#_Toc524423803)

[Proxy 20](#_Toc524423804)

[Chain of Responsibility 20](#_Toc524423805)

[Command 24](#_Toc524423806)

[Interpreter 27](#_Toc524423807)

[Iterator: 27](#_Toc524423808)

[Mediator 27](#_Toc524423809)

[Memento 27](#_Toc524423810)

[Observer 27](#_Toc524423811)

[State 27](#_Toc524423812)

[Strategy 28](#_Toc524423813)

[Template 30](#_Toc524423814)

[Visitor 32](#_Toc524423815)

[References: 32](#_Toc524423816)

# Design Pattern:

## What is Design Pattern?

In software engineering, a software design pattern is a general, reusable solution to a commonly occurring problem within a given context in software design.

A design pattern has a name, a description of the problem it addresses, and a general solution that designers must tailor to their particular variant of the problem.

## Introduction of GOF

### 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.

# Types of Design Patterns

As per the design pattern reference book **Design Patterns - Elements of Reusable Object-Oriented Software**, there are 23 design patterns which can be classified in three categories:

1. **Creational patterns**
2. **Structural patterns**
3. **Behavioral patterns**

|  |  |
| --- | --- |
| **Design Pattern** | **Description** |
| **Creational Patterns** | These design pattern deal with the object creation mechanism.  It provide more flexibility in deciding which objects need to be created for a given use case. |
| **Structural Patterns** | These design pattern deal with the relationship between classes or objects.  Concerned with changing the behavior of existing class or objects |
| **Behavioral Patterns** | These design pattern deal with the communication between objects or classes. |

# Creational Design Pattern

## What is Creational Design Pattern?

deal with the object creation mechanism

|  |  |  |
| --- | --- | --- |
|  | **Design Pattern** | **Description (GOF)** |
| **1** | **Abstract Factory** | Provide an interface for creating families of related or dependent objects without specifying their concrete classes. |
| **2** | **Builder** | Separate the construction of a complex object from its representation so that the same construction process can create different representations. |
| **3** | **Factory Method /Factory** | Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses. |
| **4** | **Prototype** | Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype. |
| **5** | **Singleton** | Ensure a class has only one instance, and provide a global point of access to it. |

# Structural Design Pattern

## What is Structural Design Pattern?

deal with the relationship between classes or objects.

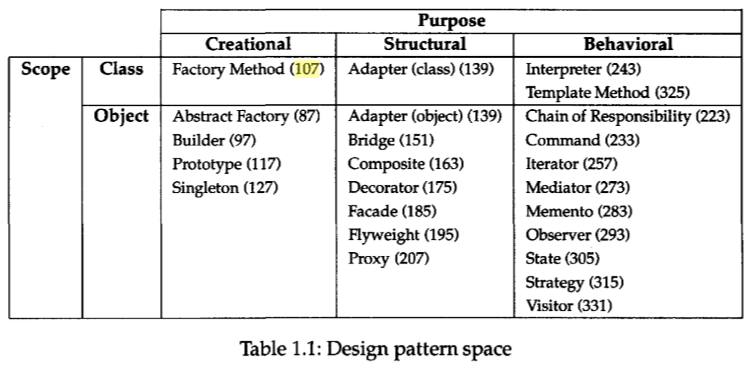
|  |  |  |
| --- | --- | --- |
|  | **Design Pattern** | **Description (GOF)** |
| **1** | **Adapter** | Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces. |
| **2** | **Bridge** | Decouple an abstraction from its implementation so that the two can vary independently. |
| **3** | **Composite** | Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly. |
| **4** | **Decorator** | Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality. |
| **5** | **Facade** | 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. |
| **6** | **Flyweight** | Use sharing to support large numbers of fine-grained objects efficiently. |
| **7** | **Proxy** | Provide a surrogate or placeholder for another object to control access to it. |

# Behavioral Design Pattern

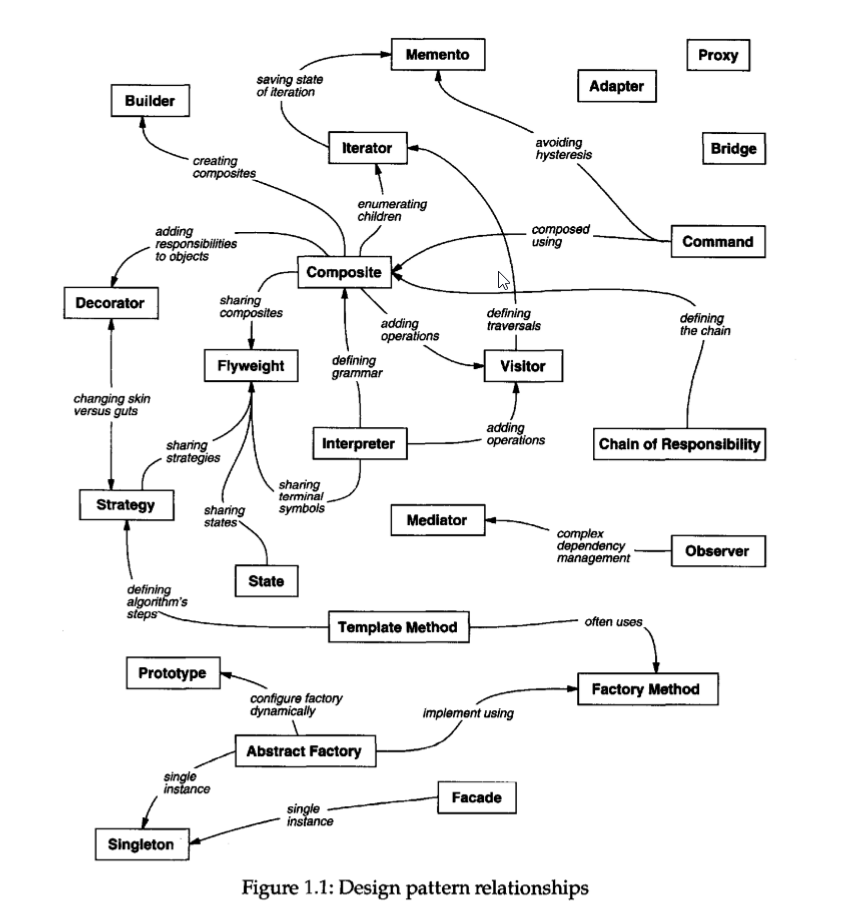
## What is Behavioral Design Pattern?

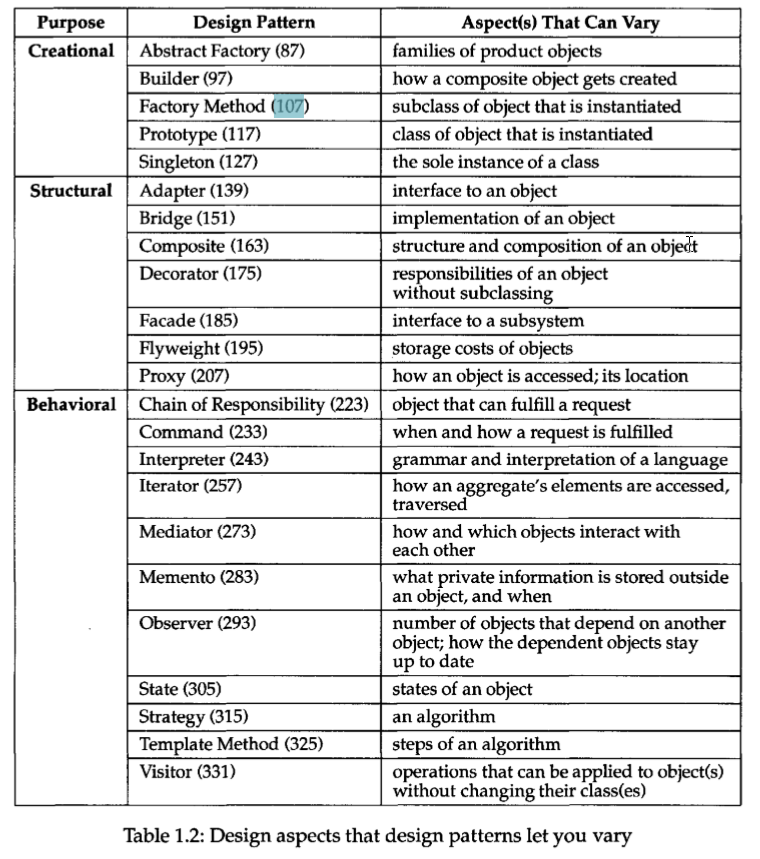
deal with the communication between objects or classes

|  |  |  |
| --- | --- | --- |
|  | **Design Pattern** | **Description** |
| **1** | **Chain of Responsibility** | Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it. |
| **2** | **Command** | Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations. |
| **3** | **Interpreter** | Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language. |
| **4** | **Iterator** | Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation. |
| **5** | **Mediator** | Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently. |
| **6** | **Memento** | Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later. |
| **7** | **Observer** | Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. |
| **8** | **State** | Allow an object to alter its behavior when its internal state changes. The object will appear to change its class. |
| **9** | **Strategy** | Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it. |
| **10** | **Template** | Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure. |
| **11** | **Visitor** | Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates. |



We classify design patterns by two criteria. The first criterion, called purpose, reflects what a pattern does. Patterns can have either creational, structural, or behavioral purpose. Creational patterns concern the process s of object creation. Structural patterns deal with the composition of classes s or objects. Behavioral patterns characterize the ways in which classes s or objects interact and distribute responsibility. The secon d criterion, called scope, specifie s whether the pattern applie s primarily to classe s or to objects. Clas s patterns deal with relationships between classe s and their subclasses . Thes e relationships are establishe d through inheritance, so they are static— fixed at compile-time. Object patterns dea l with objec t relationships , whic h can be changed at run-time and are more dynamic. Almost all patterns use inheritance to som e extent. So the only patterns labele d "clas s patterns" are those that focus on clas s relationships. Note that most patterns are in the Objec t scope.





# Abstract Factory

# Builder

**GOF:**

Separate the construction of a complex object from its representation so that the same construction process can create different representations.

**public** **class** Person {

**private** Integer id;

**private** String name;

**private** String city;

**private** **int** age;

**private** String eamil;

**private** **long** mob;

**public** Integer getId() {

**return** id;

}

**public** **void** setId(Integer id) {

**this**.id = id;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** String getCity() {

**return** city;

}

**public** **void** setCity(String city) {

**this**.city = city;

}

**public** **int** getAge() {

**return** age;

}

**public** **void** setAge(**int** age) {

**this**.age = age;

}

**public** String getEamil() {

**return** eamil;

}

**public** **void** setEamil(String eamil) {

**this**.eamil = eamil;

}

**public** **long** getMob() {

**return** mob;

}

**public** **void** setMob(**long** mob) {

**this**.mob = mob;

}

@Override

**public** String toString() {

**return** "Person [id=" + id + ", name=" + name + ", city=" + city + ", age=" + age + ", eamil=" + eamil + ", mob="

+ mob + "]";

}

}

**public** **class** PersonBuilder {

**private** Person object;

**public** PersonBuilder(){

**this**.object = **new** Person();

}

**public** PersonBuilder(Person person) {

**this**.object = person;

}

**public** Person build() {

**return** object;

}

**public** PersonBuilder setName(String name) {

**this**.object.setName(name);

**return** **this**;

}

**public** PersonBuilder setId(Integer id) {

**this**.object.setId(id);

**return** **this**;

}

**public** PersonBuilder setCity(String city) {

**this**.object.setCity(city);

**return** **this**;

}

**public** PersonBuilder setAge(**int** age) {

**this**.object.setAge(age);

**return** **this**;

}

**public** PersonBuilder setEamil(String eamil) {

**this**.object.setEamil(eamil);

**return** **this**;

}

**public** PersonBuilder setMob(**long** mob) {

**this**.object.setMob(mob);

**return** **this**;

}

}

**public** **class** PersonBuilderClient {

**public** **static** **void** main(String[] args) {

Person p1 = **new** PersonBuilder().setName("aaa").setId(1001).setAge(23).setCity("Bangalore")

.setEamil("aaa@gmail.com").setMob(7878787878l).build();

Person p2 = **new** PersonBuilder().setName("bbb").setId(1002).build();

Person p3 = **new** PersonBuilder().setName("ccc").setId(1003).setMob(89898989l).setCity("Bangalore").build();

System.***out***.println(p1);// Person [id=1001, name=aaa, city=Bangalore, age=23, eamil=aaa@gmail.com,

// mob=7878787878]

System.***out***.println(p2);// Person [id=1002, name=bbb, city=null, age=0, eamil=null, mob=0]

System.***out***.println(p3);// Person [id=1003, name=ccc, city=Bangalore, age=0, eamil=null, mob=89898989]

p2 = **new** PersonBuilder(p2).setCity("Maysore").setMob(5885858585l).build();

System.***out***.println(p2);// Person [id=1002, name=bbb, city=Maysore, age=0, eamil=null, mob=5885858585]

}

}

# Factory Method /Factory

Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

It defines an interface for creating an object, but leaves the choice of its type to the subclasses, creation being deferred at run-time.

The Factory Method pattern suggests replacing direct object creation (using a new operator) with a call to a special "factory" method. The constructor call should be moved inside that method. Objects returned by factory methods are often referred to as "products."

The participants’ classes in this pattern are:

**Product** defines the interface for objects the factory method creates.

**ConcreteProduct** implements the Product interface.

**Creator**(also refered as Factory because it creates the Product objects) declares the method FactoryMethod, which returns a Product object. May call the generating method for creating Product objects

**ConcreteCreator** overrides the generating method for creating ConcreteProduct objects



Code Example in Java:

**Product**

**publicinterface** Vehicle {

String getRequiredPrakingSpace();

}

**ConcreteProduct**

**publicclass** Bike **implements** Vehicle {

@Override

**public** String getRequiredPrakingSpace() {

**return**"Parking Space for Bike";

}

}

**publicclass** Bus **implements** Vehicle {

@Override

**public** String getRequiredPrakingSpace() {

**return**"Parking Space for Bus";

}

}

**publicclass** Car **implements** Vehicle {

@Override

**public** String getRequiredPrakingSpace() {

**return**"Parking Space for Car";

}

}

**Creator/Factory**

**1st Way**

**publicinterface** VehicleFactory {

// factory method to return a Product

Vehicle getVehicle(String vehicleType);

}

**2nd Way**

**public interface** Parking {

Vehicle getVehicle();

}

**ConcreteCreator**

**1st Way**

**public class** VehicleFactoryImpl **implements** VehicleFactory {

@Override

**public** Vehicle getVehicle(String vehicleType) {

Vehicle v = **null**;

**switch** (vehicleType) {

**case**"CAR":

**returnnew** Car();

**case**"BUS":

**returnnew** Bus();

**case**"BIKE":

**returnnew** Bike();

**default**:

**return**v;

}

}

}

**2nd Way**

**public class** BikeParking **implements** Parking {

@Override

**public** Vehicle getVehicle() {

**returnnew** Bike();

}

}

**publicclass** BusParking **implements** Parking {

@Override

**public** Vehicle getVehicle() {

**returnnew** Bus();

}

}

**publicclass** CarParking **implements** Parking {

@Override

**public** Vehicle getVehicle() {

**returnnew** Car();

}

}

**Client**

**publicclass** FactoryMethodClient {

**publicstaticvoid** main(String[] args) {

VehicleFactoryImpl vf = **new** VehicleFactoryImpl();

Vehicle v1 = vf.getVehicle("CAR");

System.***out***.println(v1.getRequiredPrakingSpace());

v1 = vf.getVehicle("BUS");

System.***out***.println(v1.getRequiredPrakingSpace());

v1 = vf.getVehicle("BIKE");

System.***out***.println(v1.getRequiredPrakingSpace());

Parking p = **new** CarParking();

System.***out***.println(p.getVehicle().getRequiredPrakingSpace());

}

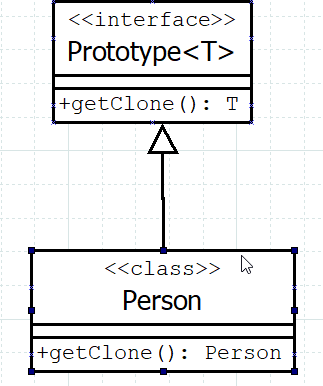
}

# Prototype

GOF: Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

* Clone the existing object instead of creating new one and can also be customized as per the requirement.

UML Diagram:



1st Approach:

**publicinterface** Prototype<T> {

T getClone();

}

**publicclass** Person **implements** Prototype<Person> {

**private** String name;

**private** String city;

**privateint**age;

**privateint**amount;

**public** Person(String name, String city, **int**age, **int**amount) {

**super**();

**this**.name = name;

**this**.city = city;

**this**.age = age;

**this**.amount = amount;

}

**public** String getName() {

**return**name;

}

**publicvoid** setName(String name) {

**this**.name = name;

}

**public** String getCity() {

**return**city;

}

**publicvoid** setCity(String city) {

**this**.city = city;

}

**publicint** getAge() {

**return**age;

}

**publicvoid** setAge(**int**age) {

**this**.age = age;

}

**publicint** getAmount() {

**return**amount;

}

**publicvoid** setAmount(**int**amount) {

**this**.amount = amount;

}

@Override

**public** String toString() {

**return**"Person [name=" + name + ", city=" + city + ", age=" + age + ", amount=" + amount + "]";

}

@Override

**public** Person getClone() {

Person p = **new** Person(**this**.name, **this**.city, **this**.age, **this**.amount);

**return**p;

}

}

2nd Approach:

**publicinterface** Prototype2<T>**extends** Cloneable {

T getClone();

}

**publicclass** Person2 **implements** Prototype2<Person2> {

**private** String name;

**private** String city;

**privateint**age;

**privateint**amount;

**public** Person2(String name, String city, **int**age, **int**amount) {

**super**();

**this**.name = name;

**this**.city = city;

**this**.age = age;

**this**.amount = amount;

}

**public** String getName() {

**return**name;

}

**publicvoid** setName(String name) {

**this**.name = name;

}

**public** String getCity() {

**return**city;

}

**publicvoid** setCity(String city) {

**this**.city = city;

}

**publicint** getAge() {

**return**age;

}

**publicvoid** setAge(**int**age) {

**this**.age = age;

}

**publicint** getAmount() {

**return**amount;

}

**publicvoid** setAmount(**int**amount) {

**this**.amount = amount;

}

@Override

**public** String toString() {

**return**"Person2 [name=" + name + ", city=" + city + ", age=" + age + ", amount=" + amount + "]";

}

@Override

**public**Person2 getClone() {

**try** {

**return** (Person2) **super**.clone();

} **catch** (CloneNotSupportedException e) {

e.printStackTrace();

}

**returnnull**;

}

}

**publicclass** PrototypeClient {

**publicstaticvoid** main(String[] args) {

Person person1 = **new** Person("abc", "Bangalore",35,5000);

System.***out***.println(person1);

Person newPerson1 = person1.getClone();

System.***out***.println(newPerson1);

Person2 person2 = **new** Person2("ikiu", "Chennai",36,6000);

System.***out***.println(person2);

Person2 newPerson2 = person2.getClone();

System.***out***.println(newPerson2);

}

}

Output:

Person [name=abc, city=Bangalore, age=35, amount=5000]

Person [name=abc, city=Bangalore, age=35, amount=5000]

Person2 [name=ikiu, city=Chennai, age=36, amount=6000]

Person2 [name=ikiu, city=Chennai, age=36, amount=6000]

# Singleton

# Adapter

GOF

Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

* Let two incompatible interfaces work together.

**The participants’ in Adapter Design Pattern are:**

1. **Adaptee**: This is the interface that needs adapting. This contains the existing functionality.
2. **Target:** New Interface which clients want to use.This is the new interface that the client wants to use. This interface’s implementation should reuse the **Adaptee** implementation (existing implementation).
3. **Adapter**: This is the implementation of the **Target** interface which reuse **Adaptee** implementation using either **HAS-A** or **IS-A** relationship.
4. **Client:** This class will interact with Adapter to use Target implementation.

UML Diagram:

Example in Java:

**Adaptee**

**public** **class** Account {

**private** **int** balance;

**public** **int** getBalance() {

**return** balance;

}

**public** **void** setBalance(**int** balance) {

**this**.balance = balance;

}

@Override

**public** String toString() {

**return** "Account [balance=" + balance + "]";

}

}

**public** **interface** PaymentGateway {

**void** doPayment(Account acc1, Account acc2);

}

**public** **class** PaymentGatewayImpl **implements** PaymentGateway {

@Override

**public** **void** doPayment(Account acc1, Account acc2) {

System.***out***.println("Payment done by Account Info!");

}

}

**Target**

**public** **class** Mobile {

**private** **long** number;

**public** **long** getNumber() {

**return** number;

}

**public** **void** setNumber(**long** number) {

**this**.number = number;

}

@Override

**public** String toString() {

**return** "Mobile [number=" + number + "]";

}

}

**public** **interface** AdvancePaymentGateway {

**void** doPayment(Mobile mob1, Mobile mob2);

}

**Adapter**

**public** **class** AdvancePaymentGatewayImpl **implements** AdvancePaymentGateway {

**private** PaymentGateway paymentGateway;

**public** AdvancePaymentGatewayImpl(PaymentGateway paymentGateway) {

**super**();

**this**.paymentGateway = paymentGateway;

}

@Override

**public** **void** doPayment(Mobile mob1, Mobile mob2) {

System.***out***.println("Payment stated using Mobile Info...");

Account acc1 = getAccountByMobile(mob1);

Account acc2 = getAccountByMobile(mob2);

**this**.paymentGateway.doPayment(acc1, acc2);

}

**private** Account getAccountByMobile(Mobile mob1) {

System.***out***.println("Get Account by Mobile..");

**return** **new** Account();

}

}

**public** **class** AdapterClient {

**public** **static** **void** main(String[] args) {

AdvancePaymentGateway apg = **new** AdvancePaymentGatewayImpl(**new** PaymentGatewayImpl());

Mobile mob1 = **new** Mobile();

Mobile mob2 = **new** Mobile();

apg.doPayment(mob1, mob2);

}

}

**Output**:

Payment stated using Mobile Info...

Get Account by Mobile..

Get Account by Mobile..

Payment done by Account Info!

# Bridge

# Composite

# 

# Decorator

GOF  
Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

The participants’ in Decorator Design Pattern are:

1. **Component:** The interface or abstract class defining the methods that has been changed dynamically.
2. **Concrete Component:** The class contains the basic implementation of the **Component** methods.
3. **Decorator:** This class contains the basic dynamic enhancement of the Component method.
4. **Concrete Decorator:** This class contains the actual dynamic enhancement of the Component method.

**UML Diagram:**

**Example in Java:**

**public** **interface** Flat {

**void** getName();

**void** getFeatures();

}

**public** **class** ModelFlat **implements** Flat {

@Override

**public** **void** getName() {

System.***out***.print("Flat in DK...");

}

@Override

**public** **void** getFeatures() {

System.***out***.print("Swimming Pool, Gym, Association, Security, 24/7 Electricity, Childern's Play Area, Cricket Play Area, Indoor Game, Park ");

}

}

**public** **class** FlatDecorator **implements** Flat {

**protected** Flat flat;

**public** FlatDecorator(Flat flat) {

**this**.flat = flat;

}

@Override

**public** **void** getName() {

**this**.flat.getName();

}

@Override

**public** **void** getFeatures() {

**this**.flat.getFeatures();

}

}

**public** **class** OneBHKFlat **extends** FlatDecorator {

**public** OneBHKFlat(Flat flat) {

**super**(flat);

}

**public** **void** getName() {

**super**.getName();

System.***out***.print(" => 1BHK");

System.***out***.println();

}

**public** **void** getFeatures() {

**super**.getFeatures();

System.***out***.print(" => One Bed Room, One Hall, One Kitchen, One Balcony, One Bathroom.");

System.***out***.println();

}

}

**public** **class** TwoBHKFlat **extends** FlatDecorator {

**public** TwoBHKFlat(Flat flat) {

**super**(flat);

}

**public** **void** getName() {

**super**.getName();

System.***out***.print(" => 2BHK");

System.***out***.println();

}

**public** **void** getFeatures() {

**super**.getFeatures();

System.***out***.print(" => Two Bed Room, One Hall, One Kitchen, Two Balcony, Two Bathroom.");

System.***out***.println();

}

}

**public** **class** DecoratorClient {

**public** **static** **void** main(String[] args) {

ModelFlat modelFlat = **new** ModelFlat();

Flat f1 = **new** OneBHKFlat(modelFlat);

f1.getName();

f1.getFeatures();

Flat f2 = **new** TwoBHKFlat(modelFlat);

f2.getName();

f2.getFeatures();

}

}

**Output:**

Flat in DK... => 1BHK

Swimming Pool, Gym, Association, Security, 24/7 Electricity, Childern's Play Area, Cricket Play Area, Indoor Game, Park => One Bed Room, One Hall, One Kitchen, One Balcony, One Bathroom.

Flat in DK... => 2BHK

Swimming Pool, Gym, Association, Security, 24/7 Electricity, Childern's Play Area, Cricket Play Area, Indoor Game, Park => Two Bed Room, One Hall, One Kitchen, Two Balcony, Two Bathroom.

# Facade:

GOF

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.

**The participants’ in Facade Design Pattern are:**

1. **Facade Interface:** This is the interface which will be exposed to the client.
2. **Services:** All services which has to be exposed by facade.
3. **Facade Client:** This will call the required service over the facade instance.

**UML Diagram:**

**Example Using Java:**

**public** **class** Department {

**private** **int** id;

**private** String name;

**private** String description;

**public** **int** getId() {

**return** id;

}

**public** **void** setId(**int** id) {

**this**.id = id;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** String getDescription() {

**return** description;

}

**public** **void** setDescription(String description) {

**this**.description = description;

}

@Override

**public** String toString() {

**return** "Department [id=" + id + ", name=" + name + ", description=" + description + "]";

}

}

**public** **interface** DepartmentService {

Department add(Department d);

Department update(Department d);

}

**public** **class** DepartmentServiceImpl **implements** DepartmentService {

@Override

**public** Department add(Department d) {

System.***out***.println("Added department : " + d);

**return** d;

}

@Override

**public** Department update(Department d) {

System.***out***.println("Updated Department : " + d);

**return** d;

}

}

**public** **class** Employee {

**private** Integer id;

**private** String name;

**private** String city;

**public** Integer getId() {

**return** id;

}

**public** **void** setId(Integer id) {

**this**.id = id;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** String getCity() {

**return** city;

}

**public** **void** setCity(String city) {

**this**.city = city;

}

@Override

**public** String toString() {

**return** "Person [id=" + id + ", name=" + name + ", city=" + city + "]";

}

}

**public** **interface** EmployeeService {

Employee add(Employee employee);

Employee update(Employee employee);

}

**public** **class** EmployeeServiceImpl **implements** EmployeeService {

@Override

**public** Employee add(Employee e) {

System.***out***.println("Added Employee : " + e);

**return** e;

}

@Override

**public** Employee update(Employee e) {

System.***out***.println("Updated Employee : " + e);

**return** e;

}

}

**public** **interface** ServiceFacade {

Department add(Department d);

Department update(Department d);

Employee add(Employee employee);

Employee update(Employee employee);

}

**public** **class** ServiceFacadeImpl **implements** ServiceFacade {

**private** EmployeeService employeeService;

**private** DepartmentService departmentService;

**public** ServiceFacadeImpl() {

**this**.employeeService = **new** EmployeeServiceImpl();

**this**.departmentService = **new** DepartmentServiceImpl();

}

@Override

**public** Department add(Department d) {

**return** **this**.departmentService.add(d);

}

@Override

**public** Department update(Department d) {

**return** **this**.departmentService.update(d);

}

@Override

**public** Employee add(Employee employee) {

**return** **this**.employeeService.add(employee);

}

@Override

**public** Employee update(Employee employee) {

**return** **this**.employeeService.update(employee);

}

}

**public** **class** FacadeClient {

**public** **static** **void** main(String... args) {

ServiceFacade sf = **new** ServiceFacadeImpl();

Department d = **new** Department();

Employee e = **new** Employee();

sf.add(d);

sf.update(d);

sf.add(e);

sf.update(e);

}

}

**Output:**

Added department : Department [id=0, name=null, description=null]

Updated Department : Department [id=0, name=null, description=null]

Added Employee : Person [id=null, name=null, city=null]

Updated Employee : Person [id=null, name=null, city=null]

# Flyweight

**According to GoF, flyweight design pattern intent is:**

Use sharing to support large numbers of fine-grained objects efficiently.

Flyweight pattern is used when we need to create a large number of similar objects.

A Flyweight object is defined

1. To store **intrinsic (invariant) state** that can be shared and
2. To provide an interface through which **extrinsic (variant) state** can be passed in.

**Intrinsic (invariant) state:**

Intrinsic state is invariant (context independent) and therefore can be shared (for example, the code of character 'A' in a given character set, this can be shared at multiple locations).

**Extrinsic (variant) state:**

Extrinsic state is variant (context dependent) and therefore cannot be shared and must be passed in (for example, the position of character 'A' in a text document, position of each text will be different so it can’t be shared).

**The participants’ in Flyweight Design Pattern are:**

# Proxy

GOF:

Provide a surrogate or placeholder for another object to control access to it.

**The participants’ in Proxy Design Pattern are:**

1. **Subject:** The interface whose method need to access with Proxy object.
2. **Concrete Subject:** The implementation of the **Subject**
3. **Proxy Subject:** The implementation of the Subject interface which create the instance of the Concrete Subject on Demand basis.
4. **Proxy Client:** The Client which use the proxy object.

**public** **interface** Account {

**void** accoutType();

}

**public** **class** SavingAccount **implements** Account{

@Override

**public** **void** accoutType() {

System.***out***.println("Saving Account");

}

}

**public** **class** ProxySavingAccount **implements** Account {

**private** SavingAccount savingAccount;

@Override

**public** **void** accoutType() {

**if** (Objects.*isNull*(savingAccount)) {

savingAccount = **new** SavingAccount();

}

System.***out***.println("Using Proxy Saving Account");

savingAccount.accoutType();

}

}

**public** **class** ProxyCLient {

**public** **static** **void** main(String[] args) {

ProxySavingAccount proxy = **new** ProxySavingAccount();

proxy.accoutType();

}

}

**Output:**

Using Proxy Saving Account

Saving Account

# Chain of Responsibility

**GOF:**

Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

* **A way of passing a request between a chain of objects.**
* Each object in the chain acts as a handler and has a successor object. If it can handle the request it does; otherwise it forwards the request to its successor.

**When not to use Chain of Responsibility:**

Don’t use chain of responsibility when each request is only handled by one handler, or when the client object knows which service object should handle the request.

**Disadvantage:**

* Execution of the request is not guaranteed, it may fall off the end of the chain if no object handles it.
* It can be hard to observe the runtime characteristics and debug.

**public** **interface** Handler {

**void** setNextHandler(Handler handler);

**void** handleRequest(Request request);

}

**public** **class** Handler100 **implements** Handler {

**private** Handler nextHandler;

@Override

**public** **void** setNextHandler(Handler handler) {

**this**.nextHandler = handler;

}

@Override

**public** **void** handleRequest(Request request) {

**if** (request.getAmount() / 100 > 0) {

System.***out***.println("Notes of 100 : " + request.getAmount() / 100);

}

**if** (request.getAmount() % 100 > 0) {

request.setAmount(request.getAmount() % 100);

nextHandler.handleRequest(request);

}

}

}

**public** **class** Handler2000 **implements** Handler {

**private** Handler nextHandler;

@Override

**public** **void** setNextHandler(Handler handler) {

**this**.nextHandler = handler;

}

@Override

**public** **void** handleRequest(Request request) {

**if** (request.getAmount() == 0 || request.getAmount() % 100 != 0) {

System.***out***.println("Entered Amount is not correct: " + request.getAmount());

**return**;

}

**if** (request.getAmount() / 2000 > 0) {

System.***out***.println("Notes of 2000 : " + request.getAmount() / 2000);

}

**if** (request.getAmount() % 2000 > 0) {

request.setAmount(request.getAmount() % 2000);

nextHandler.handleRequest(request);

}

}

}

**public** **class** Handler500 **implements** Handler {

**private** Handler nextHandler;

@Override

**public** **void** setNextHandler(Handler handler) {

**this**.nextHandler = handler;

}

@Override

**public** **void** handleRequest(Request request) {

**if** (request.getAmount() / 500 > 0) {

System.***out***.println("Notes of 500 : " + request.getAmount() / 500);

}

**if** (request.getAmount() % 500 > 0) {

request.setAmount(request.getAmount() % 500);

nextHandler.handleRequest(request);

}

}

}

**public** **class** Request {

**private** **int** amount;

**public** **int** getAmount() {

**return** amount;

}

**public** **void** setAmount(**int** amount) {

**this**.amount = amount;

}

}

**public** **class** ChainOfResponsibiltiyClient {

**public** **static** **void** main(String[] args) {

Handler handler2000 = **new** Handler2000();

Handler handler500 = **new** Handler500();

Handler handler100 = **new** Handler100();

handler2000.setNextHandler(handler500);

handler500.setNextHandler(handler100);

Request req = **new** Request();

req.setAmount(3800);

handler2000.handleRequest(req);

}

}

**Output:**

Notes of 2000 : 1

Notes of 500 : 3

Notes of 100 : 3

# Command

GOF:

Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

* A command object encapsulate a request
* Inside the request it binds together a set of actions on a specific receiver.

**Receiver** contains the action which has the operation to be performed.

**Command** contains the execute() method which call the action on receiver.

The participants’ in this pattern are:

1. **Receiver:** It contains actual action to perform.
2. **Command:** Know about which receiver to call. Each command Implementation can have separate Receiver. Encapsulate each Receiver.
3. **Invoker:** Encapsulate all Command objects.
4. **Client:** this will call a specific command at specific point. As per situation create the **Command** object and set it to **Invoker** and finally invoke() to execute() the action().

UML Diagram:

Example in Java:

**//Receiver**

**public** **interface** ElectricOperation {

**void** action();

}

**public** **class** Off **implements** ElectricOperation {

@Override

**public** **void** action() {

System.***out***.println("Switch Off");

}

}

**public** **class** On **implements** ElectricOperation {

@Override

**public** **void** action() {

System.***out***.println("Switch On");

}

}

**//Command**

**public** **interface** ElectricEquip {

**void** execute();

}

**public** **class** Fan **implements** ElectricEquip {

**private** ElectricOperation eo;

**public** Fan(ElectricOperation eo) {

**this**.eo = eo;

}

@Override

**public** **void** execute() {

System.***out***.println("Fan");

**this**.eo.action();

}

}

**public** **class** Light **implements** ElectricEquip {

**private** ElectricOperation eo;

**public** Light(ElectricOperation eo){

**this**.eo = eo;

}

@Override

**public** **void** execute() {

System.***out***.println("Light");

**this**.eo.action();

}

}

**public** **class** AirCon **implements** ElectricEquip {

**private** ElectricOperation eo;

**public** AirCon(ElectricOperation eo) {

**this**.eo = eo;

}

@Override

**public** **void** execute() {

System.***out***.println("AirCon");

**this**.eo.action();

}

}

**//Invoker**

**public** **class** SwitchBoard {

**private** ElectricEquip ee;

**public** SwitchBoard(ElectricEquip ee) {

**this**.ee = ee;

}

**public** **void** invoke() {

System.***out***.println("SwitchBoard -> Invoker");

**this**.ee.execute();

}

}

**//Command Client**

**public** **class** CommandClient {

**public** **static** **void** main(String[] args) {

SwitchBoard sb = **new** SwitchBoard(**new** Fan(**new** On()));

sb.invoke();

sb = **new** SwitchBoard(**new** AirCon(**new** Off()));

sb.invoke();

}

}

**Output:**

SwitchBoard -> Invoker

Fan

Switch On

SwitchBoard -> Invoker

AirCon

Switch Off

# Interpreter

GOF:

# Iterator:

GOF:

Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

# Mediator

GOF:

Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

# Memento

Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.

# Observer

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

# State

Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

# Strategy

GOF:

**Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.**

Encapsulates an algorithm inside a class so that it can be swapped.

The participants’ in this pattern are:

**Strategy** interface defines an action and concrete strategy classes implements the Strategy interface.

**Context** is a class which uses a Strategy.

**StrategyClient** class which set strategy object into context object and perform the action.

UML Diagram:

Example in Java:

**public** **interface** Operation {

**double** performOpertion(**int** op1, **int** op2);

}

**public** **class** AddOperation **implements** Operation {

@Override

**public** **double** performOpertion(**int** op1, **int** op2) {

**return** op1+op2;

}

}

**public** **class** SubsOperation **implements** Operation {

@Override

**public** **double** performOpertion(**int** op1, **int** op2) {

**return** op1-op2;

}

}

**public** **class** MultiOperation **implements** Operation {

@Override

**public** **double** performOpertion(**int** op1, **int** op2) {

**return** op1\*op2;

}

}

**public** **class** DivOperation **implements** Operation {

@Override

**public** **double** performOpertion(**int** op1, **int** op2) {

**return** op1/op2;

}

}

**public** **class** OperationManager {

**private** Operation operation;

**public** OperationManager(Operation operation) {

**this**.operation = operation;

}

**public** **double** calculate(**int** op1, **int** op2) {

**return** **this**.operation.performOpertion(op1, op2);

}

}

**public** **class** StrategyClient {

**public** **static** **void** main(String[] args) {

// setting Strategy Object inside the Context Object

OperationManager om = **new** OperationManager(**new** AddOperation());

// perform the action

System.***out***.println("12 + 22 = " + om.calculate(12, 22));

om = **new** OperationManager(**new** SubsOperation());

System.***out***.println("55 - 7 = " + om.calculate(55, 7));

om = **new** OperationManager(**new** MultiOperation());

System.***out***.println("44 \* 78 = " + om.calculate(44,78));

om = **new** OperationManager(**new** DivOperation());

System.***out***.println("98 / 7 = " + om.calculate(98, 7));

}

}

**Output:**

12 + 22 = 34.0

55 - 7 = 48.0

44 \* 78 = 3432.0

98 / 7 = 14.0

# Template

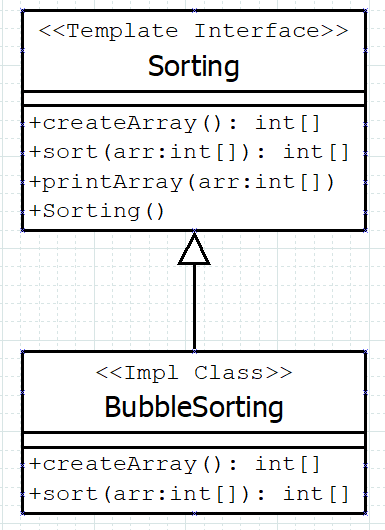
GOF:

Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

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.

UML Diagram:



**publicinterface** Sorting {

**defaultint**[] createArray() {

**int**[] arr = {11,24,3,45,7,69,9,10,7};

printArray(arr);

**return**arr;

}

**defaultvoid** printArray(**int**[] arr) {

System.***out***.println(Arrays.*toString*(arr));

}

// abstract method -> placeholder

**int**[] sort(**int**[] arr);

//template method

**defaultvoid** sorting() {

**int**[] arr = createArray();

sort(arr);

printArray(arr);

}

}

**publicclass** BubbleSorting **implements** Sorting {

@Override

**publicint**[] sort(**int**[] arr) {

**for**(**int**i=0; i<arr.length; i++) {

**for**(**int**j = 1; j<arr.length-i; j++) {

**if**(arr[j-1]>arr[j]) {

**int**temp = arr[j-1];

arr[j-1] = arr[j];

arr[j] = temp;

}

}

}

**return**arr;

}

@Override

**publicint**[] createArray() {

**int**[] arr = {11,24,3,45,7,69};

printArray(arr);

**return**arr;

}

**publicstaticvoid** main(String[] args) {

BubbleSorting bs = **new** BubbleSorting();

bs.sorting();

}

}

Output:

[11, 24, 3, 45, 7, 69]

[3, 7, 11, 24, 45, 69]

# Visitor

GOF:

Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

# References:

<http://www.w3sdesign.com/#gf>

<https://sourcemaking.com/design_patterns>

<https://en.wikipedia.org/wiki/Software_design_pattern#Classification_and_list>

<https://www.oodesign.com/>